A Node Partitioning Strategy for Optimising the Performance of XML Queries

Gerard Marks, B.Sc., M.Sc.

A Dissertation submitted in fulfilment of the
requirements for the award of
Doctor of Philosophy (Ph.D.)

to

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Mark Roantree

August 2011

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ________________ ID No.: 56213109 Date: 15/08/2011

Acknowledgements

I would like to thank all those people who made this dissertation possible. In particular, I wish to express my sincere gratitude to my supervisor Dr. Mark Roantree for his time, patience, effort and excellent guidance throughout the PhD project. Without Mark, this dissertation would not have been possible.

I would also like to thank Dr. John Murphy for recommending me to Mark as a potential PhD candidate and for his support and advice on a number of occasions during the project. I may never have completed a PhD project if it were not for John's initial encouragement.

Thanks also to Enterprise Ireland who supplied the funding for my research and to Dublin City University for the various structures and support they provided.

Thanks to my colleagues from the Interoperable Systems Group for sharing their experiences and knowledge during the time of my study. In particular, I would like to thank Martin and Jun for their help in thrashing out various ideas.

Thanks also to Paul Clarke and Donal McCann, for wading in through the December snow to do some last minute proof reading for me when everyone else was gone home for Christmas. It won't be forgotten!

Thanks to my dad, Jim, and my sister, Caroline, for all their proof reading efforts and support, it was greatly appreciated.

A very special thanks goes to my fiancée Dervla for her love, devotion, and support throughout the entire PhD project. Not only was her emotional support abundant, but her valuable opinions and her proof reading, printing, and binding skills were all exploited.

This dissertation is dedicated to Dervla and my parents.

Contents

Abstract **xi**

1 Introduction **1**

1.1 The XML Data Model 3

1.2 XML Query Processing 4

1.2.1 Performance Issues in XPath Query Processing 6

1.3 Aims and Objectives 8

1.4 Summary 10

2 Related Research **11**

2.1 Node Based Approaches 11

2.1.1 The XPath Accelerator 12

2.2 Algorithm Based Solutions 17

2.3 XML Graph Indexing 18

2.3.1 Strong DataGuides 19

2.4 Path Based Approaches 21

2.4.1 Path Indexing Approaches 21

2.5 Substituting Equijoins for Non-Equijoins 24

2.5.1 XParent and the Ancestor/Leaf Index 24

2.5.2 Proxy Indexes 26

2.6 Node Partitioning Approaches 27

2.7 Summary 29

3 **The BranchClassIndex: An Overview** **31**

3.1 Architectural Overview 31

3.2 The City Bikes XML Repository 34

4 **XML Document Partitioning** **37**

4.1 Partitioning Constructs 38

4.2 The Initial Partition Set 39

4.2.1 Initial Partitions and False Hits 41

4.3 Partition Refinement 42

4.4 Query Processing 45

4.5 Summary 48

5 **Classification of Partitions** **49**

5.1 Branch Classification 50

5.1.1 Branch Classification 51

5.1.2 Typical Build Times and Storage Costs. 54

5.1.3 Worst Case Storage Costs for BranchClassIndex . . . 56

5.2 Exploiting Branch Classification to Optimise XPath Queries . 57

5.2.1 Modelling the Indexing Constructs 59

5.2.2 Worked Example 60

5.2.3 Worked Example Summary 64

5.3 Extending the Branch Classification Process 64

5.3.1 Identifying Text Values that have Low Selectivity . . . 66

5.3.2 The Text Value Identification Algorithms 68

5.4 Post Classification Integrity 72

5.5 Branch Classification Summary 74

6 **Query Processing** **76**

6.1 Index Deployment 77

6.2 The Transformation Method 80

6.2.1 Generic Expression 82
6.2.2 Transforming the Initial XPath Step 83
6.2.3 Transforming Interim Steps 85
6.2.4 Transforming the Final XPath Step 87
6.2.5 Transforming XPath Predicate Filters 88
6.3 Sample Transformation 90
6.3.1 Transforming the Generic Expression 90
6.3.2 Transforming the Initial XPath Step 91
6.3.3 Transforming Interim Steps 91
6.3.4 Transforming the Final XPath Step 93
6.4 Index Selection 96
6.4.1 Base Index Selection Rules 96
6.4.2 Advanced Index Selection Rules 97
6.5 Summary 99
6.5.1 Integrity Checking for Transformation Process 99
6.5.2 Final Summary 100

7 Experiments 101
7.1 Evaluation Method 101
7.1.1 Implementation and Deployment Decisions 103
7.1.2 Query Categories 104
7.2 Specifying Low Selectivity Text Values 104
7.3 The Bicycle Rental Dataset 105
7.3.1 Query Analysis 106
7.3.2 Query Analysis after Text Value Classifications 114
7.3.3 Overall Query Performance 119
7.4 Comparison Using Standard Benchmarks 119
7.4.1 The XPathMark Benchmark 120
7.4.2 The Computer Science Bibliography 123
7.4.3 The Protein Sequence Database 126

7.5 Node Based Approaches 129
7.6 Summary 130

8 Conclusions **133**
8.1 Thesis Summary 133
8.2 Future Work 135
8.2.1 Reducing Redundancy 135
8.2.2 Other Future Directions 137

Bibliography **148**

List of Figures

1.1 XML Document Illustrating XDM Properties 3
1.2 XPath Twig Query Illustration 6

2.1 Pre/Post Encoding 12
2.2 The XPath Accelerator Illustrated 13
2.3 Illustrating a Strong DataGuide 19
2.4 Relational Path Based Index 22
2.5 Comparing XParent and Ancestor/Leaf 25
2.6 Partitioning factor *N=4* 28

3.1 Indexing and Query Processing Architecture 33
3.2 Single Station Sample for Lyon 36

4.1 Primary Partition Possibilities 41
4.2 Primary Partitions for Bicycle Rental Dataset 42
4.3 After Splitting Large Partitions 45
4.4 Search Space Pruning using the BranchIndex 46

5.1 XML Tree Showing Branch Classifications 52
5.2 Full Binary Tree with Distinct Path for each Node 55
5.3 ClassChain Components and Usage 58
5.4 Optimisation Constructs 59
5.5 Bicycle Rental Repository Subset 60
5.6 XML Tree Showing Branch Classifications 65

5.7 Illustrating the Effect of Low Cardinality Text Values 66
5.8 Examining Text Value Operations 67
5.9 XML Tree Showing Branch Classifications 73

6.1 XML Snippet from the Bicycle Rental Dataset 78
6.2 Node Relation and the NCLTV Index Relations 79
6.3 The XPath-to-SQL Transformation Template 81
6.4 The Completed SQL Expression 95
6.5 Index Selection 96
6.6 Index Selection (Context Shift) 98

7.1 Average Linear Path Expression Performance 108
7.2 Average for Twig Queries without Text Values 109
7.3 Average for Twig Queries with Low Selectivity Text Values . 110
7.4 Average for Twig Queries with High Selectivity Text Values . 110
7.5 Average Single Step Path Fragment Queries 111
7.6 Average Performance across all Query Categories 112
7.7 Average for Twig Queries with Low Selectivity Text Values . 115
7.8 Average Linear Path Expression Performance 116
7.9 Average for Twig Queries without Text Values 118
7.10 Average for Twig Queries with High Selectivity Text Values . 118
7.11 Average Single Step Path Fragment Queries 119
7.12 Average Performance across all Query Categories 120

8.1 Relational Index Deployment Revisited 135
8.2 Redundancy Reduction 136

List of Tables

1.1 Breakdown of an XPath query. 5

2.1 XPath Expression Breakdown 14

3.1 Overview of the System Components 34

3.2 Bicycle Rental Data Collection 35

5.1 Build Times 54

5.2 BranchCLassIndex Storage Costs 55

5.3 Extract from the Node Repository (Base Data) 61

5.4 Extract from the `NCLT` Covering Index 62

5.5 System Specification 72

5.6 Efficiency of the Text Value Identification Algorithms 72

6.1 XPath 2.0 Language Coverage 80

6.2 The Generic Expression Components 80

6.3 Populating the Class Option Variable 82

6.4 The Initial Step Expression Components 84

6.5 Populating the Class Option Variable 85

6.6 The Interim Step Expression Components 86

6.7 New Class Options 86

6.8 Equality Options for Component `C12` 87

6.9 Equality Options for Component `C13` 87

6.10 Leaf Path Expression Components 88

6.11 Components within the <*predicate-statement*> 89
6.12 Generic Expression Components 90
6.13 Initial Step Expression Components 91
6.14 First Interim Step Expression Components 92
6.15 The Second Interim Step Expression Components 93
6.16 Leaf Opening and Closing Components 94

7.1 Bicycle Rental Queries 106
7.2 Results for the Bicycle Rental Dataset 107
7.3 Branch Index Statistics 107
7.4 Bicycle Rental Dataset Results (after Text Value Classification) 113
7.5 Branch Index Statistics 115
7.6 XMark Queries 120
7.7 Results for the Mark Queries 121
7.8 BranchClassIndex Statistics (XMark) 122
7.9 Computer Science Bibliography Queries 124
7.10 Results for the Computer Science Bibliography 124
7.11 Branch Index Statistics (DBLP) 124
7.12 Protein Sequence Queries 126
7.13 Results for the Protein Sequence Database 127
7.14 Branch Index Statistics (Protein) 127
7.15 Results for the Node Based Approaches 129

Abstract

For ease of communication between heterogeneous systems, the eXtensible Markup Language (XML) has been widely adopted as a data storage format. However, XML query processing presents issues both in terms of query performance and updatability. Thus, many are choosing to shred XML data into relational databases in order to benefit from its mature technology. The problem with this approach is that (often complex and time consuming) data transformation processes are required to transform XML data to relational tables and vice versa. Additionally, many of the benefits of XML data can be lost during these processes. In this dissertation, we present a process that partitions nodes within an XML document into disjoint subsets. Briefly, as there are fewer partitions than there are nodes, a more efficient join operation can be performed between partitions, thus reducing the number of inefficient node comparisons. The number and size of partitions varies depending on the structure and layout in the XML document, and the number of partitions impacts query performance. Therefore, we also provide a partition classification process, which significantly reduces the number of partitions because each partition class represents many equivalent partitions within the XML document. In this dissertation, we will demonstrate that our approach outperforms similar approaches for a large subset of XML queries by eliminating complex join operations (where possible) during the query process.

Chapter 1

Introduction

A Markup Language can be used to annotate text with meaning. The Standard Generalised Markup Language (SGML) was adopted by the ISO in 1986 [6]. Contrary to what the name suggests, the SGML itself is not a markup language, but rather, a specification for defining markup languages. The best known application of SGML is the Hypertext Markup Language (HTML), which is used to annotate text in a way that web browsers understand.

The finite number of tags used in HTML soon became an issue because users wanted more control over web page rendering. Therefore, HTML was extended to include additional *tags* and fierce competition between *Microsoft* and *Netcape* fragmented the HTML standard. Hence, a new web page markup language was needed. At the time however, SMGL was considered too complex and therefore unsuitable for specifying the new web page markup language [6].

To overcome this issue, the eXtensible Markup Language (XML) was introduced in the late 1990's. Similar to SGML, XML is a specification language for defining markup languages. However, contrary to SGML, XML is human readable. Therefore, the development of applications that process XML data is easier. One of the first applications of XML was XHTML (a reshaped

web page development language). However, XML generated wider interest because it provided a format in which any type of data could be stored and a common format in which heterogeneous systems could communicate. For these reasons, XML is today generally accepted as the de-facto standard for information interchange.

XML data is semi-structured, which means that each datum in an XML database has its individual structure attached. This is in contrast to structured (e.g. relational) databases, where a generic structure (i.e. a schema) must be designed first, and all of the data that one wishes to store, must conform to this structure. Making changes to this generic structure, for example to insert data that has an unsuitable structure, is often a time consuming task, and it can make applications that are dependant on the data function incorrectly.

In an XML database, heterogeneous data can be inserted seamlessly because each datum has its individual structure attached, and therefore does not have to conform to a global schema. This storage flexibility has resulted in systems generating large quantities of XML data. However, as the size of XML repositories grew, the tree-centric nature of XML data resulted in significant inefficiencies in terms of query performance - especially when compared to more structured database solutions.

Three recent case studies demonstrating XML's practical usage in industry were presented by the authors of [53]. Firstly, a *Government Tax Agency* was used to demonstrate that a relational database is impractical because of *schema diversity* (one or more tables would be required for each government form). As stated in [53], this would lead to thousands of relations in a relational database as well as issues in terms of schema evolution and selecting appropriate tables for *join* operations. Secondly, an *Order Processing at a Telecommunications Company* case study showed that the mapping of diverse orders to relational schemas is difficult and scatters the details of each

order across dozens of relations; it was noted that some orders were scattered across more than 100 relations. Finally, a study of an *Event Logging at a Financial Services Company* system showed that each event is variable and application dependent. Therefore, they cannot be easily mapped to a relational database schema.

The Sensor Web is another domain that is beginning to generate large quantities of data in XML format [61]. For example, in the domain of health and human performance, XML data is generated from sensors such as heart rate monitors worn by players in team sports [50].

1.1 The XML Data Model

XML itself is not a data model, but rather a specification for defining markup languages (as discussed earlier). However, in order to perform queries across XML data it is necessary to formally specify the individual properties of an XML document. For this purpose, the W3C recommend the *XQuery and XPath Data Model* (XDM) [68].

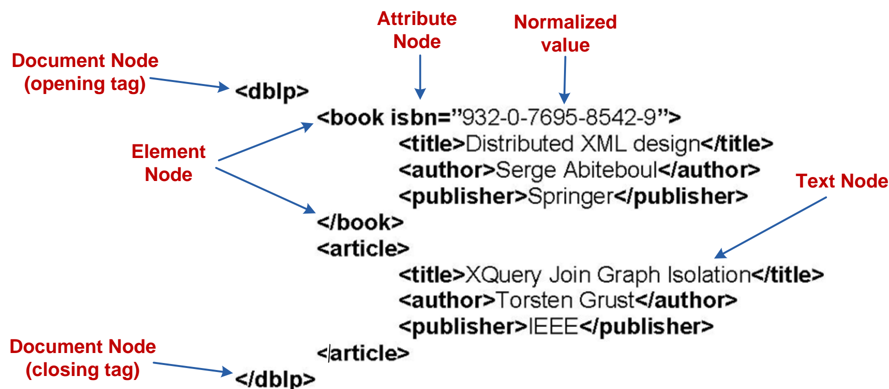


Figure 1.1: XML Document Illustrating XDM Properties

The work presented in this dissertation requires an understanding of four fundamental *node types*, which are specified in the XDM. An XML document contains a single root node called the *document node* (see Figure 1.1).

The document node contains an *opening tag* and a *closing tag* and all other nodes in an XML document will occur between these tags. The children of a document node must be *element* or *text* nodes. Other node types are permitted as children, for example *comment* nodes, but they are not relevant to this dissertation. A *text node* encapsulates XML character content. Similar to document nodes, *element nodes* have an opening and closing tag. However, there can be any number of element nodes in an XML document, whereas there is a single document node. Also, similar to a document node, an element node can have element and text nodes as its children. Unlike a document node, an element node can have one or more associated *attribute nodes*. Attribute nodes appear within an element node's opening tag. An attribute node has a *string-value*, which is the normalised value of the attribute. In this dissertation, we refer to text nodes and the normalized value of attribute nodes collectively as *text values*.

1.2 XML Query Processing

The W3C recommend two query languages, XQuery and its fundamental subset XPath, as a standard means of retrieving data from an XML database. One of the most widely documented query performance issues is associated with hierarchical relationships [3, 14, 17, 29, 34, 38, 70], that is, the time it takes to resolve parent/child and ancestor/descendant relationships. In XPath, these relationships are specific to the *ancestor*, *ancestor-or-self*, *descendant*, *descendant-or-self*, *parent* and *child* axes, which we collectively refer to as the *hierarchical XPath axes*.

As XML database systems cannot perform at the same level as their structured counterparts, many of those who rely on XML for reasons of interoperability are choosing to store XML data in relational databases rather than its native format. The advantages of semi-structured data (e.g. schemaless data storage) are therefore lost in the structured world of relational

databases, where schema design is required before data storage is permitted. The result of this is that many domains, such as sensor networks, are using rigid data models where more flexible and dynamic solutions are required. Over the last decade, many research groups have developed new levels of optimisation. However, there remains significant scope and opportunity for further improvements.

To illustrate these problems, consider the following example.

Example 1 *Retrieve the title of each masters thesis.*

XPath: */descendant::*mastersthesis*/child::*title

Step	Axis	NodeTest
1	descendant	mastersthesis
2	child	title

Table 1.1: Breakdown of an XPath query.

A *linear path expression* is an XPath query that does not contain predicate filters [28] (as illustrated in Example 1). Each linear XPath expression contains a number of *steps* (Table 1.1 shows the breakdown of the linear XPath expression in Example 1 into its steps). A step will take a sequence of nodes as input (the *context nodes*) and locate another sequence of nodes (the *target nodes*). The context node for the first step is the *document node*. The step's *axis* specifies the relationship between the context and target nodes. For example, if the axis is *descendant*, then the target nodes must be descendants of the context nodes. In other words, target nodes must be in the subtrees rooted at the context nodes. A step will also contain a *NodeTest*, which specifies the *name* the target nodes must have and their *type*, for example *element* or *attribute*.

A *twig* query is an XPath expression that contains predicate filters [28]. In other words, a linear path expression locates a subtree within the target XML document and a twig query's predicate filter(s) remove some of its

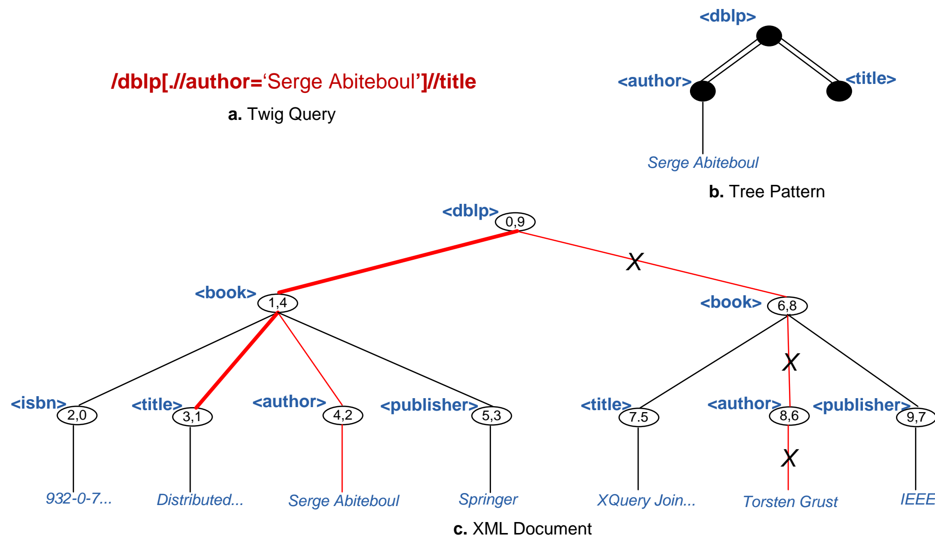


Figure 1.2: XPath Twig Query Illustration

branches. A twig query is sometimes called a *tree pattern query* [5] because the query itself can be viewed as a tree (instances of which are located with the XML document).

For example, Figure 1.2a shows a sample twig query (the predicate filters are denoted by square brackets) and 1.2b illustrates its associated tree pattern. Within the tree pattern (graph), ancestor/descendant *edges* are denoted by double lines, whereas parent/child edges are shown as single lines. Figure 1.2c shows the tree representation an the XML document. The red edges show the paths that the twig query specifies and the thicker red edges are the path to the target node. The path marked with an 'X' are branches that failed to satisfy the predicate filter.

1.2.1 Performance Issues in XPath Query Processing

Sequence-oriented [30] evaluation of XPath steps is inefficient as all of the nodes in the sequence of context nodes must be compared, based on an XPath *axis*, to all of the nodes in the sequence of target nodes. Initially, the sequence of target nodes contains all of the nodes in the XML document

(we refer to them as the *initial target nodes*). In a naive system, the entire sequence of target nodes will be traversed once for each context node or vice versa, which, as we will show, is inefficient.

Another issue with sequence-oriented evaluation of XPath steps is *duplication of work*. Duplication of work occurs when the 'regions' (of an XML document) 'associated with the' (XPath) 'step are evaluated independently for each context node' [32]; we refer to this as *node-at-a-time* evaluation of XPath steps. For example, based on the descendant axis, a single target node may be a descendant of multiple context nodes. If this occurs, a node-at-a-time processor will traverse the target node multiple times. As we will discuss shortly, *node-at-a-time* evaluation of XPath steps has a query performance overhead that is often unnecessarily large. The fundamental objective of an XML query processor is to reduce the number of nodes that must be visited during query processing.

Current XML query optimisation solutions can be placed in two broad categories. On one hand, *index-based* approaches build indexes on XML documents to provide efficient access to data. Index-based solutions, for example, XRel [69], XPath Accelerator [29], Xeek [42], benefit from existing *join* algorithms such as those that are available to standard Relational Database Management Systems (for example, *NestedLoops*, *HashJoin*, *MergeJoin*). Index-based approaches can also exploit mature relational facilities, such as *Cost Based Optimisers*, to select appropriate *query execution plans* based on specific XML data and queries. On the other hand, *algorithm-based* (or *Native* XML) approaches are focused on designing new join algorithms, for example, TJFast [38], StaircaseJoin [32], which are specifically designed to support queries across XML data.

The *XPath Accelerator* [29] is a node-at-a-time XPath query processor and it demonstrated that an XPath index stored inside a relational database can be used to evaluate all of the XPath axes. However, the XPath Accelerator

suffers from the aforementioned issues associated with node-at-a-time processors, such as *duplication of work*. Thus, the XPath Accelerator suffers from significant scalability issues as noted in [42].

Path-based approaches [24, 25, 34, 69] avoid visiting many nodes during the query process by storing each node's *root path* [28] (the path from the document node to itself) in a path-index (sometimes called a *path-summary* index [7]). As many node instances can share the same root path there are usually much fewer root paths than there are nodes in the XML document. An XPath expression can be divided into multiple path fragments such as *primary path fragments* as specified in [24]. Node-at-a-time comparisons are only required between primary path fragments. Thus, if primary path fragments span more than one step in an XPath expression (which they often do), inefficient node-at-a-time comparisons are not required at every step in an XPath expression (unlike the node based approaches).

In a different approach, a special type of *node partitioning* allows nodes of different *names* and *types* (element/attribute) to reside in the same partition [43]. The motivation to do this is based on the fact that there will always be fewer partitions than there are nodes in the XML document. Thus, the partitions that contain the target nodes can be identified more efficiently and after the relevant partitions are identified, only the nodes that comprise these partitions need to be visited using costly node-at-a-time evaluations. However, to the best of our knowledge, [43] is the only such index-based node partitioning approach and it requires a *user defined* partitioning factor for each XML document. Thus, the user must run time-consuming experiments to identify a suitable partitioning factor for each XML document.

1.3 Aims and Objectives

The hypothesis put forward in this research is that larger numbers of node-at-a-time comparisons can be avoided during the query process through *node*

partitioning and *partition classification*. Furthermore, unlike the approach presented in [43], the time-consuming preprocessing stage that is used to identify suitable partitioning factors can be avoided. Finally, a node partitioning approach can be independent of particular XML node labels. This allows the most suitable XML encoding scheme to be chosen based on the user's needs, while exploiting the performance benefits of the approach presented in this dissertation. To demonstrate the effectiveness of our approach, standard query performance benchmarks are used to compare this approach to that of other researchers and XML database vendor systems. In addition, data and queries, taken from a real world application which generates large quantities of sensor data in XML format are used to demonstrate the wider applicability of our approach. The main objectives of our research can be highlighted as follows:

- To provide a novel partitioning method for XML data storage that offers improved levels of optimisation for XML queries.
- To develop efficient algorithms that automatically identify and resize document partitions. This is unlike the existing approach that requires a preprocessing phase, which is infeasible for large XML documents (we present experiments to substantiate this claim).
- To exploit structural information to allow identical node partitions to be merged and thus, reduce the size of the index and avoid processing large numbers of equivalent node partitions during the query process.
- To provide a relational deployment of the indexing structures that encourages relational query optimisers to choose efficient *query execution plans*.
- To provide an *XPath-to-SQL* transformation process that produces SQL queries that are engineered specifically for the approach presented in this dissertation.

1.4 Summary

In this Chapter, a general introduction to XML and XML query processing was provided. XML has been widely adopted as a data storage format because XML data does not have to conform to a generic schema. In addition, XML databases can evolve easily without time-consuming schema re-design. XML also provides a common format in which heterogeneous systems can communicate, which has led to an explosive growth in its usage and the size of XML repositories as a whole.

A major obstacle to improving XML query performance is the tree-centric nature of the data and in this Chapter we discussed how node-at-a-time XPath step evaluation is not scalable. Thus, an XPath query optimiser's primary objective is to visit as few nodes as possible during the query process. There are two main approaches to fulfilling this objective: (1) index-based approaches materialise data structures in advance of query processing to support query optimisation, (2) algorithm-based approaches bypass nodes by making decisions during query processing itself.

We begin in Chapter 2 by examining related solutions to XML query performance; in Chapter 3 an architectural overview of the system is provided and we introduce a real world XML case study; in Chapter 4, we provide a detailed description of the data structures used in the partitioned index and provide a step by step description of how it is built; in Chapter 5, we present the classification process for node partitions, which reduces the number of partitions while maintaining the same degree of search space pruning, and supports an additional node bypassing mechanism; in Chapter 6, we discuss index deployment and query processing; in Chapter 7, we present our experiments and discuss the findings; finally, Chapter 8 presents conclusions and future work.

Chapter 2

Related Research

There are several approaches to XPath query optimisation. In Chapter 1, we broadly categorised these efforts into *index-based* and *algorithm-based* approaches. Early index-based approaches to XPath query optimisation are presented in §2.1 and their inefficiencies are identified. Following this in §2.2, we provide an evaluation of how algorithm-based approaches overcome some of these shortcomings before introducing more advanced index-based approaches.

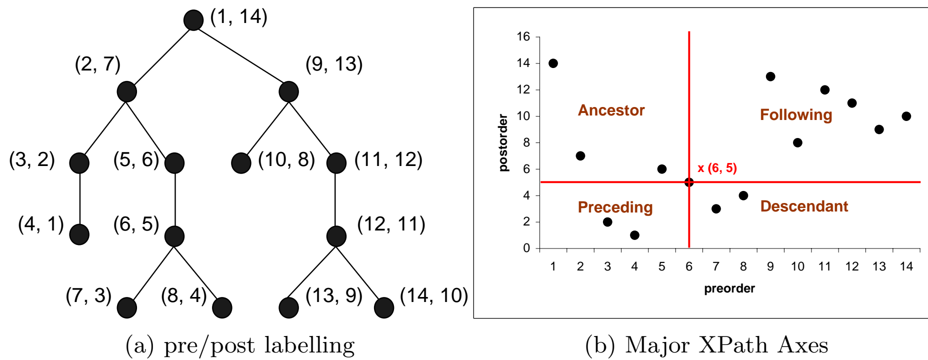
In §2.3, XML schema graph indexing is introduced as its concepts are used throughout the remaining index-based approaches; in §2.4, path-based indexes are evaluated; §2.5 discusses approaches that convert *non-equijoins* to more efficient *equijoins* to optimise node-at-a-time evaluations; finally, node partitioning approaches are discussed in §2.6.

2.1 Node Based Approaches

Node based approaches are those that do not use structure, (i.e. DataGuides, XML schemas, or Document Type Definitions), to optimise XPath steps. These approaches evaluate the XPath axes by comparing individual node labels in which the relationships between nodes are encoded.

2.1.1 The XPath Accelerator

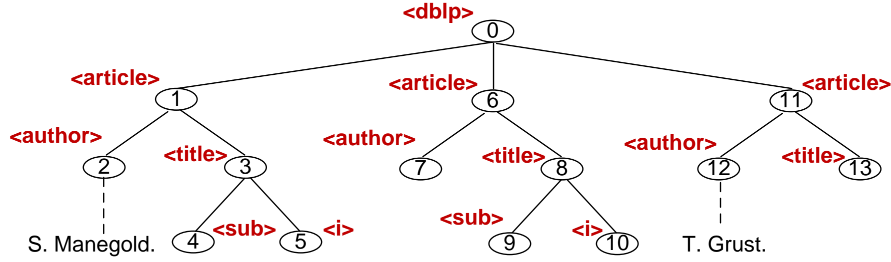
The *XPath Accelerator* [29] is an XML index which is designed for deployment in relational databases. In this work, *pre/post* labels (known as *Dietz* encoding [22]), are used to encode each node with the region of the XML document that it encompasses. From the context of any given node, *pre/post* labels can be used to partition all other nodes in the XML document into its *ancestor*, *descendant*, *following* and *proceeding* nodes, that is, the four *major XPath axes*. Figure 2.1a depicts an XML tree that is labelled with (pre/post) identifiers and 2.1b illustrates how those nodes are dispersed in the pre/post plane. In particular, notice the four major XPath axes associated with node `x` (6,5). For example, the nodes that are ancestors of node `x` will have a preorder identifier that is less than `6` and a postorder identifier greater than `5`. The other major axes can be resolved using similar pre/post logic.



(a) pre/post labelling (b) Major XPath Axes

Figure 2.1: Pre/Post Encoding

Figure 2.2 illustrates how a source XML document (2.2a) is stored in the node relation (2.2b). The first two columns in the node relation contain the *preorder* and *postorder* identifier of each node respectively. The major XPath axes can be sub-partitioned or extended (to include the context node itself) using the *minor XPath axes*. The most common of these are the *parent* and *child* axes, which sub-partition the ancestor and descendant axes



a. (Source XML Dataset)

pre	post	par	att	tag	cdata
0	13	null	no	dblp	null
1	4	0	no	article	null
2	0	1	no	author	null
3	3	1	no	title	S. Manegold.
4	1	3	no	sub	null
5	2	3	no	i	null
6	9	0	no	article	null
7	5	6	no	author	null
8	8	6	no	title	null
9	6	8	no	sub	null
10	7	8	no	i	null
11	12	0	no	article	null
12	10	11	no	author	T. Grust.
13	11	11	no	title	null

b. (Node Relation for XPath Accelerator)

Figure 2.2: The XPath Accelerator Illustrated

respectively. For the purpose of evaluating the parent and child axes, the XPath Accelerator assigns an additional *par* (parent) label to each node. The *yes or no* (boolean) column *att* is used for differentiating between *attribute* and *element* nodes as specified in the *XQuery and XPath Data Model* (XDM) [68]. Node label *tag* is the *name* of the element or attribute node. Finally, *cdata* is used to associate each node with its text value, i.e. text values associated with element or attribute nodes.

Example 2 *(Sample XPath Expression)*

//article[./author = 'T. Grust.']/descendant::title.

The step-at-a-time XPath evaluation process that is used by the XPath Accelerator is referred to many times throughout this dissertation, thus we

Step	Axis	Name	Type	Predicates
1	descendant-or-self	article	element	child::author = 'T. Grust.'
2	descendant	title	element	

Table 2.1: XPath Expression Breakdown

will now spend some time detailing it. The XPath expression in Example 2 specifies: *find the title of Torsten Grust's articles*. The expression contains two steps and the breakdown of each step is shown in Table 2.1. Each step in an XPath expression receives a sequence of *context nodes* and locates another sequence of nodes, which we call the *target nodes*. The sequence of target nodes initially contains all of the nodes in the XML document. Then the sequence of *initial target nodes* is reduced to the *actual target nodes* based on the following:

- The step's axis relative to the context nodes. For example, if the *descendant* axis is specified, the target nodes must be descendants of the context nodes.
- The node's *name* (*article*) and *type* (element/attribute) to satisfy the step's NodeTest.
- The text value of element or attribute nodes (T. Grust.).

The sequence of context nodes for the first step in an *absolute* XPath expression is always a single node sequence containing the *document node*. The document node is always assigned the first preorder and last postorder value, for example, node (`0, 13`) in Figure 2.2b. For all subsequent steps, the context nodes are the *actual target nodes* that were located at the previous step. Using the query in Example 2, the process is as follows:

1. Find the *element* nodes (we know it is an element node because attribute node names in an XPath expression have the prefix '@') called *article* that are descendants of the *document node*. As all nodes are

descendant of the document node, in the first step all nodes that have *tag* 'article' and *att* 'no' are returned from the node relation (Figure 2.2b), the sequence (1, 6, 11). For brevity, the sequence here contains the preorder identifiers only.

- The predicate filter, denoted by *square brackets*, in the first step specifies that: *only the articles that were written by Torsten Grust should be returned.* The first (and only) step in the predicate receives the sequence of context nodes: (1, 6, 11). The step's axis is child, denoted by the abbreviated syntax '/'. The NodeTest specifies that the target nodes must have the *tag* '*author*', *att* '*no*' and *cdata* '*T. Grust.*' and they must be children of at least one of the context nodes. In this instance, the *par* label is exploited to return those nodes that have the *par* equal to `1`, `6` or `11`. In other words, all nodes whose parent is a context node are returned. The target node sequence is (`12`). Thus, as this predicate is a filter on the first step in the XPath expression, all of the target nodes in the first step that do not have node `12` as a child are filtered out. Therefore the sequence of target nodes is reduced to (`11`).

2. The final step receives the context node sequence (`11`) and the step's axis is descendant. The NodeTest specifies that only nodes that have *tag* '*title*' and *att* '*no*' are returned. This time the preorder and postorder labels are used to ensure that the target nodes are descendants of node `11`, that is, the nodes that have a preorder label greater than that of node `11`, and a postorder label less than that of node `11`. See *pre/post* ranges in Figure 2.1 for more details. Thus, the node sequence (`13`) is returned.

In [31], a variant of the XPath Accelerator is presented, in that it uses

pre/size/level labels instead of *pre/post/par*. The *level* label replaces the *par* label used by the XPath Accelerator to evaluate the parent and child axes. The benefit of *pre/size/level* is that the *size* variant of *post* minimises the overhead of *node relabelling* upon updates to the XML document, as *size* is invariant with respect to subtree copying or moving, whereas *post* is not [64].

For the purpose of query performance however, [31] describes how *partitioned B-trees* (i.e. multi-column B-tree indexes) and *context pruning* can be used to optimise the performance of XPath queries in standard Relational Database Management Systems (RDBMSs). In this approach, partitioned B-trees are used to optimise non-recursive XPath axes (parent and child). Optimisation is achieved by minimising *false hits*; in other words, avoiding nodes that cannot contribute to the result. For example, a partitioned B-tree index on columns: *(level,pre)* allows the relational *query optimiser* to avoid false hits at levels that cannot contribute to the result. Additionally, as columns such as *level* and *type* have only a small number of possible values (that is, they have *low selectivity*), partitioned B-trees that are prefixed with these columns have lower creation and maintenance overheads than those prefixed on columns that have *high selectivity* [31].

Critical Evaluation

While the XPath Accelerator has the benefit of supporting all of the XPath axes, *range comparisons* (e.g. pre/post comparisons) were shown to be highly inefficient for large XML documents [40–42]. The experiments presented in this dissertation show that even when pushed to their limit using the techniques described in [31], node based approaches are inefficient because they perform too many node-at-a-time comparisons. The remaining approaches discussed in this Chapter describe ways of reducing the number of node comparisons that must be performed, therefore improving query performance.

2.2 Algorithm Based Solutions

The MPMGJN (Multi-Predicate MerGe JoiN) [70] algorithm is similar to the standard *merge join* algorithm used for equijoins in a relational database, but it is tailored to evaluate structural joins efficiently. A merge-join algorithm performs a join between two lists of nodes. Two cursors are created, one pointing to the head of each list. The cursors are compared to each other as they advance forward to perform the join operation. The MPMGJN differs to the standard merge-join by skipping nodes as the cursors are advanced forward [28].

In [3], it was noted that the MPMGJN could not evaluate the '/' (child) axis efficiently in certain circumstances as it visits descendant nodes that are not actually child nodes. In [3], a StackTree is proposed which (unlike MPMGJN) avoids processing the unnecessary descendant nodes, thus improving performance. As the name suggests, StackTree uses a stack structure to store nodes that are nested on the same path in data trees. It was also shown in [63] that the MPMGJN and StackTree could be optimised further by calculating partitions of the *pre/post* plane that can be avoided, thus reducing the number of nodes that are evaluated.

The MPMGJN, StackTree, and similar approaches [32, 63], are *binary* join algorithms. A binary join in the case of XPath query processing is a join based on a sequence of *context nodes* and a sequence of *initial target nodes* at each step in the XPath expression. In other words, a binary join is a join between lists of nodes. This is exactly the approach used by the node based approaches detailed in §2.1. It soon became apparent that the binary join approach processed large numbers of *intermediate* nodes that could be avoided if more than two XPath steps are evaluated simultaneously.

PathStack and TwigStack use a more holistic approach to perform structural joins unlike the binary join algorithms. In contrast to StackTree, PathStack and TwigStack use multiple stacks to cache nested nodes. Each node in

a stack has a pointer to its corresponding node in its parent stack which enables the maintenance of possible *n-ary* path solutions. Other advancements in holistic twig pattern matching solutions include TwigStackList [37], TwigList [59] and those that are based on prefix labelling schemes (much like Dewey decimal) TJFast [38], Twig2Stack [17].

Critical Evaluation

Relational database technology is a mature technology and thus, there are many advantages to indexing XML data within a relational database such as mature query optimisation and transaction management technology [30, 62]. In addition, modern query optimisers that are available in relational databases, such as the *cost based optimiser* in Oracle 11g, are well suited for choosing suitable query execution plans for XML queries (if the XML index is well designed).

An issue with algorithm-based approaches (as identified in [31]) is that they require significant modifications to the relational database kernel (which is the approach used by [11]). Alternatively, *Native* XML databases [23,33,51] can incorporate algorithm-based approaches to facilitate query optimisation.

2.3 XML Graph Indexing

Each datum in an XML database has its structure attached, which is the fundamental difference between semi-structured and structured data (structured data must conform to a global structure). Thus, structural information can be extracted from an XML dataset. In this section, we discuss various graph indexing schemes that exploit this structural information for the purpose of query processing.

2.3.1 Strong DataGuides

XML graph indexing is based on the concept of a *Strong DataGuide* [27]. Strong DataGuides are defined in terms of a graph that is not required to be a *tree*. When the graph is a tree, as is the case of an XML dataset, a Strong DataGuide reduces to what is known as a 1-index [28,52]. Therefore, we will now introduce the concept of Strong DataGuide in the form that is relevant to an XML tree, that is, as defined for a 1-index [52].

In a 1-index, a node in the source dataset (a *data node*) maps to a single distinct node in the index (an *index node*), that is, many nodes in the base data map to the same index node. One or more data nodes map to the same index node if they are *B-bisimilar* (backward bisimilar) [28]. Two data nodes are B-bisimilar if they have the same *root path*; the path that contains their *name* and the name of each of their ancestors in root-to-leaf order. Figure 2.3a shows a source dataset (taken from the *Computer Science Bibliography* [21]) containing data nodes. The 1-index is depicted in Figure 2.3b.

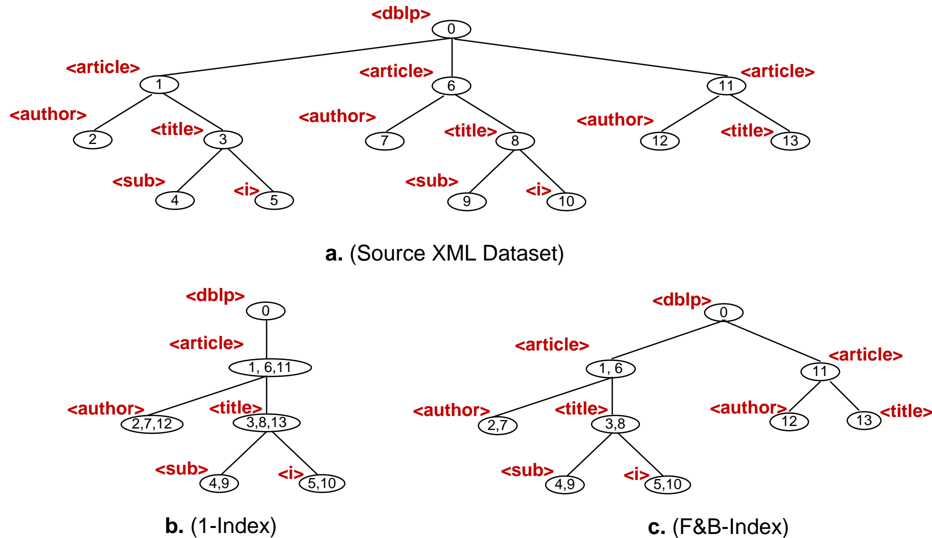


Figure 2.3: Illustrating a Strong DataGuide

Critical Evaluation

There are two major issues with the 1-index. Firstly, the size of the 1-index can be equal to the size of the XML document in the worst case and even in situations where it is smaller than the size of the original dataset, it is usually too large to be efficient [28]. Similar approaches [16, 19, 36] have minimised the size of the 1-index by shortening the number of nodes in the root path. For example, by shortening the root path, the A(k) index [36] trades index size for query accuracy, i.e. the result set for the query may contain unwanted nodes, but the target nodes are definitely in the result set. However, more important is the fact the 1-index and similar approaches can only satisfy *linear path queries*, i.e. they cannot satisfy Twig queries [28].

It is widely thought that the smallest index that can satisfy all Twig queries is an F&B (Forward and Backward bisimilar) Index [1, 35, 65]. An F&B Index requires not only that the incoming root path is bisimilar, but also that the node's outgoing paths be bisimilar. This is illustrated in Figure 2.3c. Notice how node `11` in the 1-index (Figure 2.3b) is separated from nodes `1` and `6` in the F&B Index. This is because node `11` has different outgoing paths than nodes `1` and `6`. Similar is true for node `13`.

Index graph approaches such as the 1-index cannot evaluate Twig queries, thus they provide coverage of a very small subset of the XPath language. Although an F&B Index can evaluate Twig queries, it is typically too large to be used in practice [28, 35, 65].

One approach [35] tried to minimise this problem by creating multiple F&B Indexes, each of which satisfies a subset of Twig queries. However, a more significant issue is that schema graph index structures are not suitable for indexing in a relational database [28, 39]. For this reason, (and because of the size of 1-indexes, F&B indexes and similar approaches), *path-based* indexing structures that prune the search space for Twig queries (rather than evaluating them) have become popular. Therefore, we will now discuss

path-based approaches.

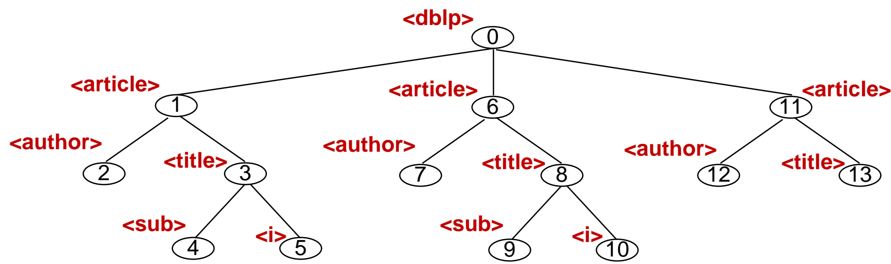
2.4 Path Based Approaches

As discussed in §2.3, schema graph indexes are generally too large in practice. For this reason, many approaches exploit the same structural information as schema graph indexes, but for the simpler task of *search space pruning*. We have categorised this class of XPath index as *path-based* indexes.
The main similarity between path-based approaches and the graph indexing schemes (discussed in §2.3) is that the *root path* of each node is stored in a separate *path index* (sometimes referred to as *path summary*). Each distinct root path can only occur once in the path index and many nodes in the base data map to the same root path in the path index. Thus, regular path expressions can be executed across the path index to prune search space for linear path expressions. Additionally, Twig queries can often be evaluated as multiple linear path expressions.

2.4.1 Path Indexing Approaches

XRel [69] is a relational implementation of a path-based index. Figure 2.4 illustrates how a source XML dataset (2.4a) can be indexed in a relational database (2.4b and 2.4c). In XRel, each distinct *root path* (as described earlier for the 1-index) is assigned an identifier and is stored along with its path in a relation as shown in Figure 2.4c. Each node in the node relation (as shown in Figure 2.4b) is stored along with its path identifier. In other words, there is a many-to-one mapping from nodes in the node relation to their root path in the path relation.

Example 3 ***(Linear XPath Expression)***
/dblp/article/title.



a. (Source XML Dataset)

Pre	Name	Type	PathID
0	dblp	element	p1
1	article	element	p2
2	author	element	p3
3	title	element	p4
4	sub	element	p5
5	i	element	p6
6	article	element	p2
7	author	element	p3
8	title	element	p4
9	sub	element	p5
10	i	element	p6
11	article	element	p2
12	author	element	p3
13	title	element	p4

b. (Node Relation)

PathID	Path
p1	dblp
p2	dblp/article
p3	dblp/article/author
p4	dblp/article/title
p5	dblp/article/title/sub
p6	dblp/article/title/i

c. (Path Relation)

Figure 2.4: Relational Path Based Index

XRel divides a given XPath expression into one or more *simple path expressions* (similar to *regular expressions*). These simple path expressions can be executed across the path relation using the `LIKE` facility in the relational database. The `LIKE` keyword in SQL syntax allows regular path expressions to be executed across columns that contain character strings, such as the *Path* column in Figure 2.4c. Thus, the path identifiers for each simple path expression can be identified in the Path relation. After path identifiers have been determined, a *join* predicated on these path identifiers in the node relation will return all nodes associated with the simple path expression. Example 3 shows an XPath expression. In the Path relation, this XPath expression is mapped to the path identifier `p4`. All of the nodes in the node relation that have the path identifier `p4` can be identified (nodes {3, 8, 13}).

The search space is pruned because those nodes that do not have the path identifier `p4` are avoided.

The authors of [24] show how a path-index (identical to that of XRel) can be exploited to evaluate multiple XPath steps in both forward axes (descendant, child) and backward axes (ancestor, parent) directions. This means that multiple contiguous descendant and child steps, which are called *path fragments*, can be evaluated simultaneously, as can contiguous ancestor and parent path fragments. In contrast, XRel can only evaluate path fragments that contain the child and descendant axes [24].

Example 4 ***(XPath Twig Query)***
/dblp//title[./sub].

Critical Evaluation

There are a number of inefficiencies associated with path-based approaches. Firstly, if the XPath expression is a Twig query as in Example 4, the expression must be separated into multiple simple path expressions. In Example 4 there are two simple path expressions, these are: `/dblp//title` and `/dblp//title/sub`. In this instance, the path relation is used to determine the path identifiers associated with each of the path expressions in turn. The nodes associated with each of the path identifiers are located in the node relation resulting in two separate node *sets*. Inefficient node-at-a-time evaluations are then required between these two *node sets*. Earlier in this Chapter, we explained why this type of node evaluation is inefficient.

As discussed in §2.1, traditional node based approaches must perform this type of inefficient join once for each step in the XPath expression. Thus, path-based approaches provide query performance benefits over traditional node based approaches as there will be fewer joins when there are fewer simple paths than there are steps in the query expression. However, the experiments described in Chapter 7 show that large numbers of these node

comparisons are often required. In fact, we identify a category of XPath queries in which these approaches cannot provide any optimisation.

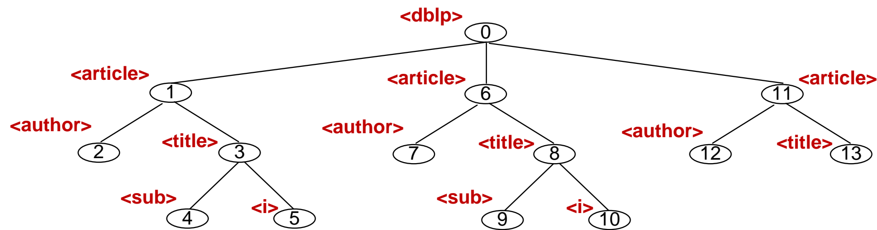
According to the authors of [28], another inefficiency inherent in these approaches is that SQL can support exact string matching (across path indexes) efficiently using *equijoins* and B^{+}-trees tree indexes on strings. However, they cannot efficiently support the regular expressions that are required to evaluate XPath's *descendant* or *ancestor* axes. Additionally, [28] noted that regular path expressions associated with path-based approaches can produce incorrect results when recursion exists in the XML data. Reversed path approaches [18, 56] minimise these issues by reversing and then encoding the root paths associated with each node, but we believe that the much greater issue is the performance of inefficient join operations between path fragments (or their encoded alternatives).

2.5 Substituting Equijoins for Non-Equijoins

As discussed, the largest inefficiency associated with node based and path-based approaches is related node-at-a-time comparisons. These node comparisons are usually based on node ranges such as those in the pre/post plane. The authors of [40–42] pointed out that much of the inefficiency is a result of using non-equijoins to perform these range comparisons. In this section, a number of approaches are presented that substitute non-equijoins with more efficient equijoins where possible.

2.5.1 XParent and the Ancestor/Leaf Index

To optimise XPath axes, XParent [34] proposed that the *transitive closures* of the XML tree can be precomputed and stored. In other words, one map from each node to its *set* of ancestor nodes and a second map to its *set* of descendant nodes are pre-materialised in the index. XParent uses a



a. (Source XML Dataset)

pre	post	par	att	tag	cdata
0	13	null	no	dblp	null
1	4	0	no	article	null
2	0	1	no	author	null
3	3	1	no	title	S. Manegold.
4	1	3	no	sub	null
5	2	3	no	i	null
6	9	0	no	article	null
7	5	6	no	author	null
8	8	6	no	title	null
9	6	8	no	sub	null
10	7	8	no	i	null
11	12	0	no	article	null
12	10	11	no	author	T. Grust.
13	11	11	no	title	null

a. Node Relation (Base Data)

a	d
0	0
1	0
1	1
2	0
2	1
2	2
3	0
3	1
3	3
4	0
4	1
4	3
4	4
5	0
5	1
5	3
5	5
6	0
6	6
7	0
7	6
7	7
8	0
8	6
8	8
9	0
9	6
9	8
9	9
10	0
10	6
10	8
10	10
11	0
11	11
12	0
12	11
12	12
13	0
13	11
13	13

b. XParent

a	d
2	0
1	1
2	2
4	0
4	1
4	3
4	4
5	0
5	1
5	3
5	5
7	0
7	6
7	7
9	0
9	6
9	8
9	9
10	0
10	6
10	8
10	10
12	0
12	11
12	12
13	0
13	11
13	13

c. Ancestor/Leaf

Figure 2.5: Comparing XParent and Ancestor/Leaf

relational database to deploy the index, therefore the transitive closures are stored in a *relation* as shown in Figure 2.5(b); in this relation, the left column contains ancestor nodes, and the right column contains their descendants. Query performance is gained because the ancestor and descendant XPath axes can be evaluated using equijoins across this relation, which are more efficient than non-equijoins [18, 42] (range comparisons); range comparisons are used in the node-based approaches described earlier.

However, the size of the transitive closures is typically too large to be used in practice [28]. This is because many nodes in an XML document share the same ancestors and descendants, and thus, the XParent approach leads to a large amount of duplicated (redundant) storage.

The Ancestor/Leaf index [41] provides a more compact equijoin evaluation strategy. In this instance, each *leaf* node is stored along with its ancestors (in contrast to every node being stored with its ancestors and descendants) as depicted in Figure 2.5(c). This leads to less duplicated storage but since many leaf nodes share common ancestors, the storage costs of this approach are still significant.

Critical Evaluation

In practice, XParent and the Ancestor/Leaf indexes often incur prohibitive storage costs. However, it was shown that the Ancestor/Leaf outperforms the node-based approaches [40], which validates the use of equijoins over non-equijoins where possible. For this reason, the approach presented in this dissertation uses equijoins rather than non-equijoins for mapping between branch partition classes described later in Chapter 6.

2.5.2 Proxy Indexes

ProxyReach [41] focuses on reducing the duplicated storage resulting from the XParent and Ancestor/Leaf indexes. In this instance, a single proxy

node can be selected anywhere on the root-to-leaf path in an XML tree to represent a larger group of nodes. The proxy node is then stored along with all its ancestors and descendants. For example, if the proxy node is specified to be every leaf node, then the proxy index will be identical to the Ancestor/Leaf index. However, if proxy nodes are selected higher on the path, then the proxy can represent multiple root-to-leaf paths; thus, decreasing the storage requirements. Xeek [42] is a variant of ProxyReach in that it only requires that the ancestors of each proxy node are stored, thus further reducing the storage requirements.

Critical Evaluation

While proxy indexes have a benefit where storage minimisation is crucial, there is a trade-off between query performance and storage requirements. The reason for this is that many of the ancestors and descendants of proxy nodes represent false hits for the XPath step. In fact, the higher on the root-to-leaf path that proxy nodes are selected, the higher the number of false hits, that is, there will be fewer false hits in the Ancestor/Leaf index. These false hits must be removed using costly node comparisons such as those described for node based approaches earlier. In the case of these proxy indexes, *pre/post* labels are used (as with the XPath Accelerator). In summary, proxy indexes reduce storage requirements but to the detriment of query performance.

2.6 Node Partitioning Approaches

Node partitioning approaches segment documents into disjoint subsets. As there are fewer partitions than nodes in an XML dataset, a more efficient join operation can be performed between partitions, which reduces the workload for the more costly task of node comparisons. To the best of our knowledge,

[43] is the only major index-based approach in this area.

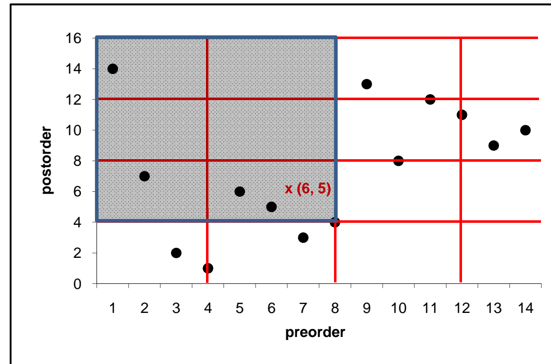


Figure 2.6: Partitioning factor *N=4*

In [43], the *pre/post* plane is partitioned based on a user defined *partitioning factor*. Figure 2.6 illustrates the *pre/post* plane partitioned into *parts* using a partitioning factor of 4. For each node, the *pre/post* identifier of its *part* is the *lower bound* of its x and y values respectively. For example, in Figure 2.6, the *part* P associated with node $x(6, 5)$ is $P(4, 4)$. The ancestors of node x can only exist in the *parts* that have a lower bound x value ≤ 4 and a lower bound $y \geq 4$, that is, the shaded parts (Figure 2.6). Similar is true for the other major XPath axes, for example, descendant, following and preceding.

Critical Evaluation

The problem with this approach is that an ideal partitioning factor is not known in advance and requires rigorous experimentation to identify. For example, in reported experiments each XML document was evaluated for the partitioning factors 1, 2, 4 up to 256 [43]. This type of experimentation is infeasible even for relatively small XML documents. Additionally, as XML data is often irregular, uniform partitioning of all nodes in an XML document based on a single partitioning factor may not be optimal. Finally, although it is suggested in [43] that the partitioning approach may

be tailored to other encoding schemes such as *order/size*, it relies heavily on the *lower bound* of each x and y value in the partitioned *pre/post* (or order/size) plane. Therefore, this approach does not naturally lend itself to *prefix* based encoding schemes such as [8, 10, 55], which have become popular in recent years because they facilitate updates more easily than region encoding approaches such as *pre/post* (discussed earlier in this Chapter).

2.7 Summary

In this Chapter, we have discussed related work in the area of XPath query optimisation. Initially, traditional node-at-a-time query evaluation strategies were discussed with their inefficiencies highlighted. We then discussed approaches that eliminate these inefficiencies.

Native XML join algorithms (*binary* and *n-ary*) were then evaluated and their benefits discussed. We stated the reasons why these approaches cannot be deployed in a relational database without significant changes to its kernel, which suggests that an index-based approach that performs at the same level as their native counterparts may be a more preferable solution. For this reason, the remainder of the Chapter was dedicated to index-based solutions that can be deployed in a relational database.

We presented graph indexing schemes (for example, 1-index, F&B index) and described the subsets of the XPath language that they cover. We showed that these approaches often result in an index that is too large to be efficient. Also, it was stated that these approaches are not suitable for deployment in a relational database, but their concepts are used in path-based approaches which can be deployed in a relational database.

In terms of search space pruning, we described how path-based approaches provide significant performance gains over step-at-a-time processing of XPath expressions. However, these approaches often require inefficient node-at-a-time comparisons between path fragments, which are costly.

Between individual steps in an XPath expression or between path fragments, XParent and similar approaches show how inefficient non-equijoins can be converted to more efficient equijoins by explicitly storing the transitive closures of nodes. However, we described why the size of the transitive closures is often too large to be used in practice. Additionally, some approaches that reduce the size of the transitive closures were introduced, but it was shown that they are either still too large to be used in practice or they trade query performance for reduced storage costs.

Finally, we showed an alternative approach that uses disjoint node partitions. As there are fewer partitions than there are nodes, the partitions that contain the target nodes can be identified more efficiently, which reduces the number of costly node-at-a-time comparisons. However, this approach often requires significant preprocessing to identify suitable partitioning factors and it is heavily dependent on the properties of a single encoding scheme, which limits its possible application areas.

Chapter 3

The BranchClassIndex: An Overview

This chapter begins by introducing the architecture of the entire XML indexing and query processing system. The goal of this chapter is to provide a high level overview of the different steps in XML document processing and querying. Thus, it will provide the reader with a brief description of each of the system's components.

While we benchmark our work against that of others using standard XML datasets and queries, we focus our attempts at query performance on a real world dataset to demonstrate the wider impact of our work. In §3.2, we provide an introduction to this real world XML dataset (the City Bikes XML repository).

3.1 Architectural Overview

In this section, a description of each of the processes is provided. Our objective is to familiarise the reader with each of these. We begin with an explanation of the overall system, which details the system's components and the order in which each process and indexing construct is created and

executed, respectively.

The overall architecture is illustrated in Figure 3.1. Initially, an XML document (d1) is received by the partitioning process (p2). Thus, the partitioning process creates the index of node partitions: the *branch index* (i3). Process p2 also identifies the properties of each node, such as *name*, *type* and *level*, which serve as input to the process that creates the *node repository* (p5). The node repository (sometimes referred to as the *base data* [35]) contains an entry for each node in the XML document. In other words, it is a complete representation of the original XML tree.

Due to the fine granularity of our partitioning process, the branch index will, for many XML documents, be too large. In other words, the branch index is exploited to prune the search space for queries, but searching the branch index provides an undesirable performance overhead. Therefore, we compact the branch index using a branch classification process (p4). The output of this classification process is the *branch class index* (i9) in which a single branch class represents many branch instances - the classification process (p4) is the focus of Chapter 5. The class index is a compact version of the branch index.

Using the properties of each node that is received from the partitioning process (p2) and the branch class information received from the classification process (p4), process p5 generates the node repository (i6). Process (p7) subsequently generates the `NCLT` (Name/Class/Level/Type) index (i8). As the node repository is deployed in a relational database, the `NCLT` index can be generated using simple SQL expressions. Process (p7) generates the `NCLT` index (i8) by selecting distinct *name*, *class*, *level*, and *type* from the node repository. Later, we will show how, based on the specific properties of our partitioning process, the `NCLT` index can act as a *covering index* [35] for XPath steps that contain hierarchical axes. A covering index contains all of the attributes that are required by the query (or sub-query in this instance),

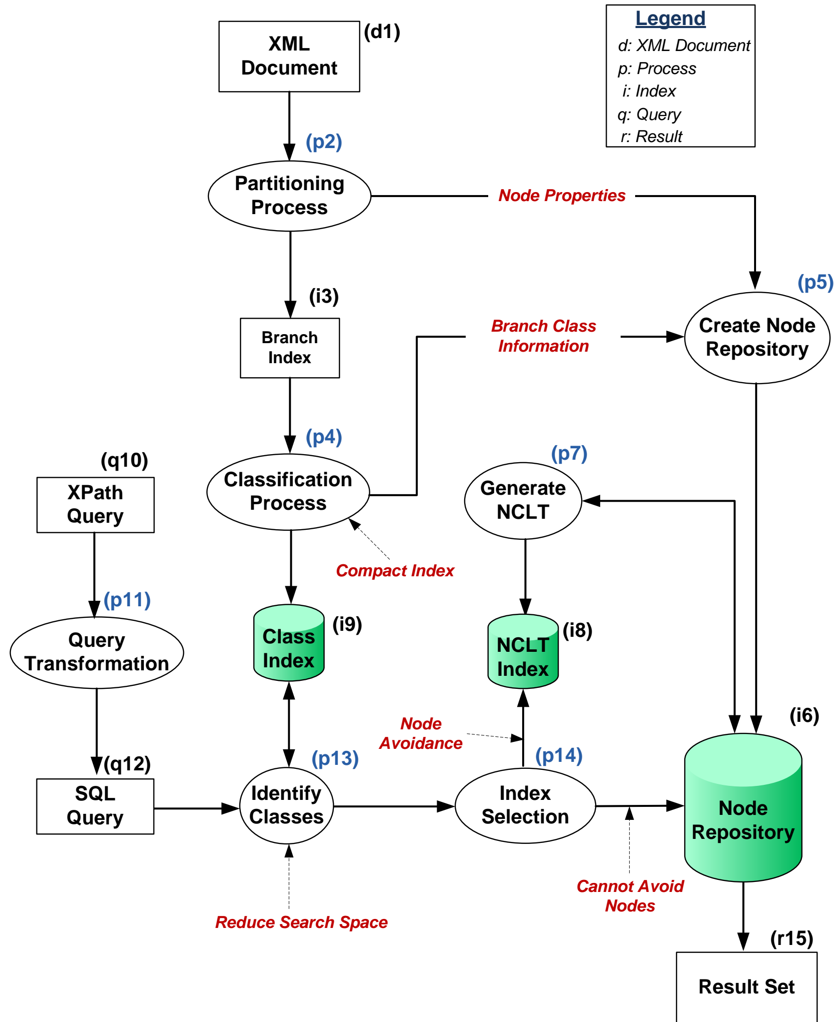


Figure 3.1: Indexing and Query Processing Architecture

therefore the entire query can be evaluated in the index. In [31], they created a covering index for XPath steps by exploiting *partitioned B-tree* facility in a standard relational database [31] (as discussed in Chapter 2). In contrast, the `NCLT` index is a covering index that is specifically designed to optimise the hierarchical XPath axes and the improvement achieved using the `NCLT` index is substantial as we will show in Chapter 7.

Upon receiving an XPath query (`q10`), the *XPath-to-SQL* transformation process (`p11`) transforms the XPath query to its SQL equivalent (`q12`). The

SQL corresponding to each *step* in the XPath expression is an SQL sub-expression. Each of these sub-expressions in turn uses the class index (i9) to reduce the search space. The index selection process (p14) then chooses when to query the high performance NCLT index (i8) where possible, or else the node repository (i6) will be selected. Finally, when each sub-clause is complete, the result set (r15) is returned.

Name	**Chapter**
Partitioning Process (p2)	Chapter 4
Classification Process (p4)	Chapter 5
Node Repository Creation Process (p5)	Chapters 4 5
Generate the NCLT (Name/Class/Level/Type) Index (p7)	Chapter 5
XPath-to-SQL Transformation Process (p11)	Chapter 6
Branch Class Identification Process (p13)	Chapter 6
Index Selection Process (p14)	Chapter 6

Table 3.1: Overview of the System Components

Table 3.1 provides a summary of the system's processes and indicates the chapters in which more details can be found.

3.2 The City Bikes XML Repository

The use of sensors in the physical-world is constantly increasing and could now be regarded as widespread. The number of applications built on top of such sensor data is also increasing. Examples are urban traffic watch; weather monitoring; tracking of goods.
Recently the city of Dublin, like many other European cities, deployed a bike sharing scheme in which the public can rent (and return) a bike from stations located throughout the city centre. Stations are equipped with sensors that monitor bike availability and publish such data to the DublinBikes website (*www.dublinbikes.ie*). Users can connect to the website (through a PC or mobile application) to check where stations are, how many bikes are available, how many spaces are available to return bikes, and what type of

payment methods are available.

Using this data providers can understand at which station it is better to pick up or return bikes for maintenance in order to minimise service disruption. In effect, the web service offers an efficient mechanism for determining the current status of bike or space availability.

There are many situations in which it is advantages to be able to access historical data, or look for trends and patterns over time. For example, city planners or the companies offering the rental service must determine the location for new sites; determine those sites that require expansion; or reduce or close sites that are unpopular. Furthermore, this analysis must take place over time to avoid any bias that could result from poor weather patterns or other varying factors.

City	Country	Stations	Data Size
Aix-en-Provence	France	16	8 KB
Amiens	France	25	8 KB
Besancon	France	30	8 KB
Lyon	France	340	80 KB
Mulhouse	France	35	12 KB
Nancy	France	25	8 KB
Nantes	France	89	24 KB
Plaine-Commune	France	44	12 KB
Rouen	France	18	8 KB
Dublin	Ireland	40	12 KB
Toyama	Japan	16	8 KB
Luxembourg	Luxembourg	46	12 KB
Santander	Spain	13	4 KB

Table 3.2: Bicycle Rental Data Collection

The bicycle rental application [49] records information on bike availability in cities and towns across the world. The data is collected from each location at regular intervals (see Table 3.2) and the dataset at the time of our experiments (Chapter 7) was 2.06 GB in size.

A typical XML document for a single station in the city of Lyon (France) is shown in Figure 3.2. It shows that an entry was taken from Lyon on the `01/06/2010` at `19:59:50`. The entry also illustrates the weather conditions and the unit of measurement. Finally, for each bicycle station, information

```
<bikes>
  <city>
    <Lyon day='01' month='06' year='2010'>
      <stations>
        <time>
          <hour>19</hour>
          <minute>59</minute>
          <second>50</second>
        </time>
        <timeOfDay>19:59:50 01-06-2010</timeOfDay>
        <timeUnit>milliseconds</timeUnit>
        <timeStart>1275418790000</timeStart>
        <weather>
          <time>Tue, 01 Jun 2010 8:30 pm CEST</time>
          <wind>
            <chill>63</chill>
            <direction>40</direction>
            <speed unit="mph">7</speed>
          </wind>
          <humidity>59</humidity>
          <pressure unit="inches">29.97</pressure>
          <temp unit="degrees farenheit">63</temp>
          <condition>Partly Cloudy</condition>
          <weatherTimeTaken>75</weatherTimeTaken>
        </weather>
        <station>
          <id>9052</id>
          <timeTaken>2853</timeTaken>
          <available>2</available>
          <free>20</free>
          <total>22</total>
          <ticket>1</ticket>
          <error>0</error>
        </station>
      </stations>
    </lyon>
  </city>
</bikes>
```


Figure 3.2: Single Station Sample for Lyon

related to availability such as the number of *free* spaces and *available* bikes is provided. A segment of the bicycle rental dataset will be used to provide examples throughout the remainder of this dissertation.

Chapter 4

XML Document Partitioning

The motivation for document partitioning is that there will be fewer partitions in the XML document than there are nodes. Thus, the partitions that contain *target nodes* can be identified more efficiently; and the nodes that comprise all of the other partitions are eliminated from the search space. In §4.1, the new constructs that are used in the node partitioning process are introduced. This is followed in §4.2, with a step-by-step description of how the initial node partitions are created. The initial partitions are disjoint *sets* of nodes, which collectively contain every node in the XML document. However, we will show that the initial partitions act merely as a platform for XML optimisation and can be improved by avoiding unnecessary *false hits* [31]. Thus, the partitioning process is updated in §4.3 to create partitions of more desirable sizes with respect to the hierarchical XPath axes. In §4.4, a description of the query process for each of the hierarchical XPath axes is provided. Finally in §4.5, a summary of the concepts introduced in this chapter is provided.

4.1 Partitioning Constructs

This section introduces constructs that form part of the partitioning process. In [48], we defined a disjoint partition of nodes within an XML document as a *branch*. A branch construct and its sub-types are now described - illustrated examples of these constructs will follow in §4.2 and §4.3.

Definition 1 *A branch is a* set *of connected node identifiers within an XML document, where node identifiers are unique proxies for nodes in an index, for example pre/post labels.*

A branch is the abstract data type used to describe a partition of nodes. In our work, we will deal with the local-branch and path-branch sub-types of a branch.

Definition 2 *A local-branch is a branch, such that its members represent a single branching node and the nodes in its subtree. A local-branch cannot contain a member that represents a descendant of another branching node.*

The local-branch uses the branching node to form each partition. Our process uses the rule that each local-branch must not contain nodes that are descendants of another branching node to create primary partitions.

Definition 3 *A path-branch is a branch with a single path.*

The path-branch is an abstract type with no branching node. Each member is a child member of the preceding node. Its three sub-types (*orphan-path*, *branchlink-path* and *leaf-path*) are used to partition the document.

Definition 4 *An orphan-path is a path-branch such that its members cannot belong to a local-branch.*

The orphan-path definition implies that members of the orphan-path cannot have an ancestor that is a branching node. The motivation is to ensure that each node in the XML document is now a member of some partition.

Definition 5 *A branchlink-path is a path-branch that contains a link to a single descendant partition of its local-branch.*

In any local-branch, there is always a single branching node and a set of non-branching nodes. With the non-branching nodes, we must identify those that share descendant relationships with other partitions. These are referred to as branchlink-path partitions and each member occupies the path linking two branching nodes (i.e. two partitions).

Definition 6 *A leaf-path is a path-branch that contains a leaf node inside its local-branch.*

A leaf-path differs from a branchlink-path in that it does not contain a link to descendants partitions. In other words, it contains a single leaf node and its ancestors.

4.2 The Initial Partition Set

In the first attempt at partitioning, the goal is to include all nodes in local-branch or path-branch partitions. Path-branches are abstract types and at this point, all path-branch instances will be orphan-paths. Throughout this section, we describe the partitioning process with respect to *element* nodes. In all cases, *attribute* and *text* nodes in an XML dataset are placed in the same branch as their associated element node. This results in fewer branches which can be exploited to boost query process.

The algorithms for encoding an XML document using a *pre/post* encoding scheme were provided by the authors of [29]. In brief, each time a *starting tag* is encountered a new *object* is instantiated with the following attributes of an element node: *name, type, level,* and *preorder*. Subsequently, the new element object is pushed onto a *stack* structure: the *element stack*. Each time an *end tag* is encountered an element is popped from the element stack and is assigned a postorder identifier.

Once an element has been popped from the stack, we call it the *current node*, and the *waiting list* is a *set* in which elements reside temporarily prior to being indexed. The first step in the process is to determine if the current node is a branching node by checking if it has more than one child node. The next steps are as follows:

1. If the current node is non-branching and does not reside at *level 1* (one level greater than the level at which the document node resides), it is placed on the *waiting list* until step three (below).

2. If the current node is branching, it is assigned to the next local-branch in sequence. Also, the nodes on the waiting list that are its descendants are placed in the same local-branch and are removed from the waiting list.

3. If the current node is non-branching, but a node at *level 1* is encountered, the current node does not have a branching node ancestor. Therefore, the current node is assigned to an *orphan-path*. For the same reason, any node currently on the waiting list is assigned to the same orphan-path.

At the end of this process, only the *document node* is unassigned. As the document node is a generic ancestor of all other nodes in the XML document, it can be ignored during the partitioning process. Indexing of the document node is described as part of the final index structure in Chapter 5. Figure 4.1 illustrates the set of local-branches `LB-1` to `LB-8` and orphan-paths `OP-9` and `OP-10`.

Figure 4.1: Primary Partition Possibilities

4.2.1 Initial Partitions and False Hits

While Figure 4.1 shows the layout of branch instances in a hypothetical situation, we will now illustrate the layout of the primary partitions using a small segment from the real world bicycle rental dataset that was introduced in Chapter 3. The authors of [31] described *false hits* as the visiting of unnecessary nodes during the query process. In particular, they showed how false hits can be reduced by avoiding nodes at specific *levels* in the XML document that cannot contain target nodes. We will now illustrate the problem of *false hits* relative to the primary partitions and their impact on query performance before demonstrating (in §4.3) how a refinement of the primary partitions leads to fewer false hits, and thus, improves query performance.

Figure 4.2 illustrates the primary partitions for a small segment of the bicycle rental dataset. In this instance, there are four local-branches and no orphan-paths. To illustrate the concept of *false hits*, we will analyse the query process for the *ancestor* axis. Given a sequence of context nodes, we must identify the branch instances that the context nodes belong to, and their ancestor branches. If the sequence of context nodes contains nodes `11` and `12` (Figure 4.2), local-branches `LB-2` and `LB-4` will be identified. The search space is pruned at this point because nodes `5`, `6`, `7`, `8`, `18`, `19`, `20` and `21` do not reside in `LB-2` or `LB-4` and thus, are not visited during query processing. However, there are only three nodes within local-branch `LB-4` that can be hierarchically related to context nodes from `LB-4`, these are nodes `1`, `9` and `10`. This means that nodes `2`, `3`, `4`, `15`, `16`, and `17` are *guaranteed* false hits. False hits can be eliminated using individual node comparisons (e.g. using *pre/post* labels), but this is inefficient (as discussed in Chapter 2).

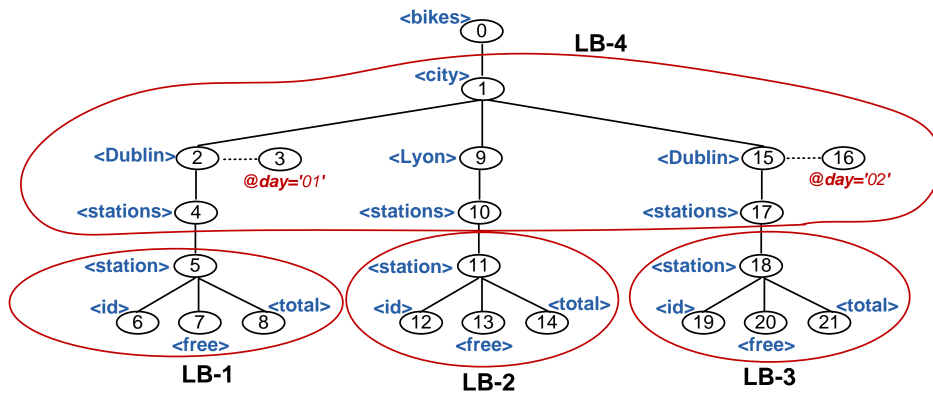


Figure 4.2: Primary Partitions for Bicycle Rental Dataset

4.3 Partition Refinement

We have identified a principle that minimises the number of false hits: *only hierarchically related nodes are permitted in each branch* and local-branches can contain nodes that do not have a hierarchical association (i.e. an ances-

tor/descendant relationship). When nodes within branches do not share a hierarchical association, the false hits previously described will occur (causing inefficiencies for the hierarchical XPath axes).

Each local-branch instance has a single branching node *root* which may have many (non-branching node) descendants. It is the non-branching descendants of the root that are examined to determine if they share a hierarchical association as we want to ensure that only nodes that are hierarchically related reside in the same partition. Thus, we partition the non-branching nodes (in each local-branch) into disjoint path-branches (Definition 3). As orphan-paths and local-branches are *disjoint*, each of these path-branch instances will be a *branchlink-path* or a *leaf-path*.

The `RefinePartitions` (Algorithm 1) replaces all steps outlined for creating the primary index (above). The new branch partitions are created by processing *two* local-branches simultaneously. All *current nodes* (see creating primary partitions above) up to and including the first branching node are placed in the first *waiting list* (*wList1*) where they *wait* to be indexed. Subsequently, the next set of current nodes, up to and including the second branching node, are placed on the second waiting list (*wList2*). At this point, *wList1* and *wList2* contain the nodes that comprise the first and second local-branches respectively.

If a node at *level 1* is encountered, the nodes that comprise *wList2* are an orphan-path (*line 2*). If a *branchlink-path* exists, `RefinePartitions` identifies it as the non-branching nodes in *wList2* that are *ancestors* of the root node in *wList1* (*lines 6-7*). If one or more *leaf-paths* exist, they will be the nodes in *wList2* that are not ancestors of root node in *wList1* (*lines 8-9*). The remaining nodes that comprise the first local-branch (*wList1*) are then moved to the index (*line 12*); this will be the *single* branching node root of the first local-branch only. At this point, the only node that remains in *wList2* is the root node of the second local-branch. This local-branch

Algorithm 1 `RefinePartitions`

```
1: if node at level 1 encountered then
2:    move nodes that comprise wList2 to orphan-path;
3: end if
4: move non-branching nodes from wList1 to leaf-path;
5: for each node n in wList2 do
6:    if n = ancestor of wList1.ROOT ∧ n ≠ branching node then
7:       move n to branchlink-path;
8:    else if n ≠ ancestor wList1.ROOT then
9:       move n to leaf-path;
10:   end if
11: end for
12: move local-branch from wList1 to local-branch;
13: move local-branch from wList2 to wList1;
```

is then *moved* to *wList1* (*line 13*) and thus, *wList2* is emptied. The next local-branch is placed in *wList2* and the process is repeated until all nodes are processed. When this process has completed, the result will be many more partitions, with the benefit of increased pruning.

A second function of this process is to track and index the *ancestor-descendant* relationships between branch partitions. This is achieved by maintaining the *parent-child* mappings between branches. Given two branch instances: `B1` and `B2`, `B2` is a child of `B1` *if and only if* the *parent* node of a node that is in `B2` belongs to `B1`. When the `RefinePartitions` process is complete, the *ancestor-descendant* relationships between branches are determined using a recursive function across these parent-child relationships, that is, by selecting branch's children, then its children's children recursively.

The layout of the final branch possibilities is illustrated in Figure 4.3. In particular, notice that the nodes within each branch instance are hierarchically related and branching nodes always end up in a single node branch, which is important for the query process, which we describe next.

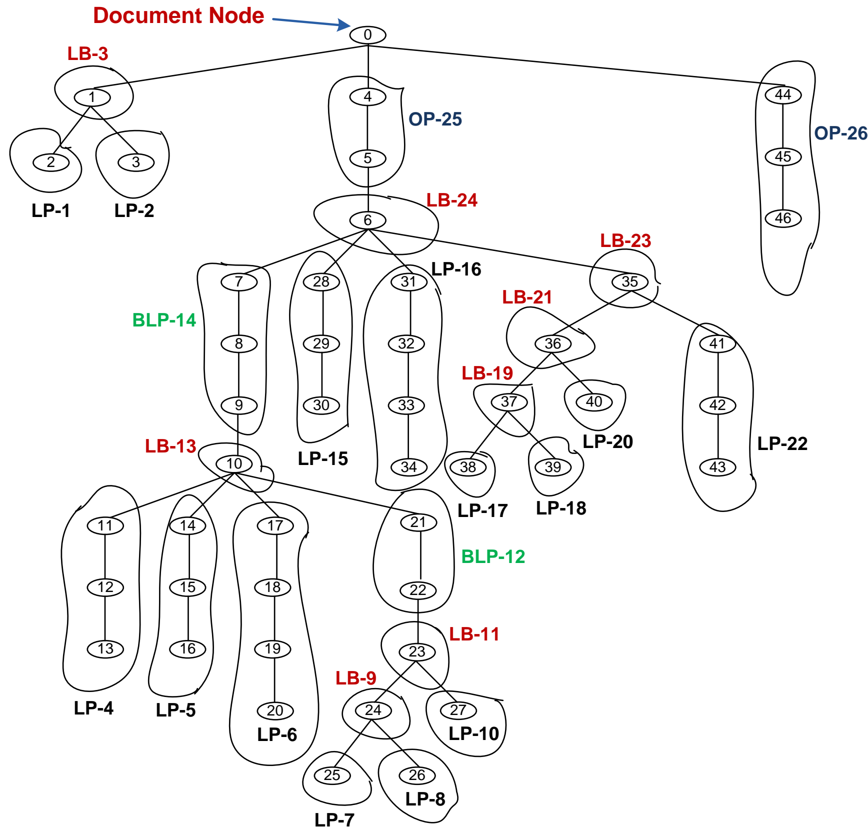


Figure 4.3: After Splitting Large Partitions

4.4 Query Processing

The branch index (which we refer to as *BranchIndex*) has some very useful properties which can be exploited for the purpose of query performance. This is illustrated in Figure 4.4. Notice how the BranchIndex is exploited to prune the search space by limiting the region of the node repository that must be evaluated. Moreover, these properties are fundamental to the approach described in the subsequent sections and are thus explained here. Consider the six hierarchical XPath axes: *ancestor, ancestor-or-self, descendant, descendant-or-self, parent* and *child*. We described in Chapter 2 how a step in an XPath expression receives a sequence of context nodes. Then, from an initial sequence of target nodes (the entire XML document), the

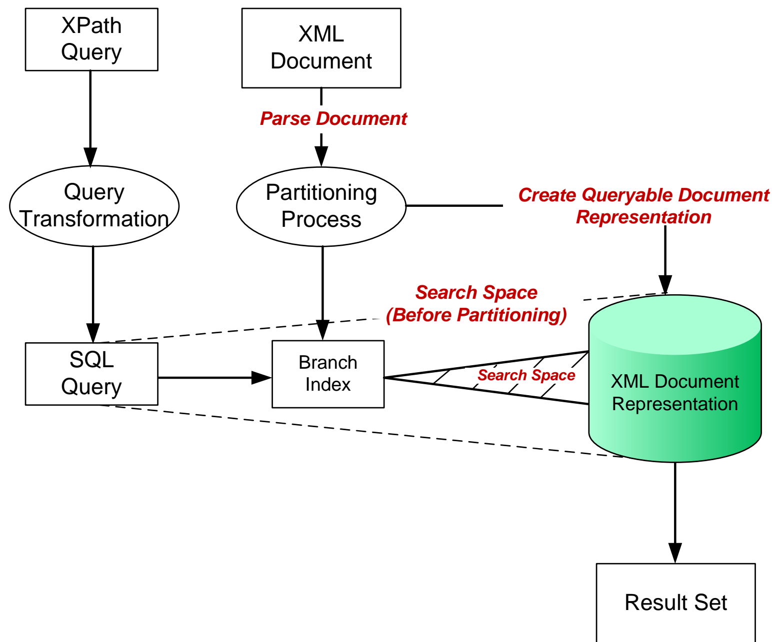


Figure 4.4: Search Space Pruning using the BranchIndex

sequence of actual target nodes are located by performing a *join* between the two sequences. This *join* operation is based on the step's *axis*, *NodeTest* and *predicates*. In our approach, we achieve performance improvements by firstly (before the *join* operation) identifying the *set* of branch instances associated with the sequence of context nodes. The branch instances that are identified are based on the step's *axis* as follows:

- *ancestor, ancestor-or-self axes.* Identify the **ancestor-or-self** branches associated with each context node.

- *descendant, descendant-or-self axes.* Identify the **descendant-or-self** branches associated with each context node.

- *parent axis.* Identify the **parent-or-self** branches associated with each context node.

- *child axis.* Identify the **child-or-self** branches associated with each context node.

In each case, the *self* branch is the branch in which the context node resides and the remaining branches are those that are hierarchically related based on the step's axis. Thus, the identified branches will *always* contain the *target nodes*. It is important to note that, after the identification of the branch instances associated with the sequence of context nodes, a node's *level* attribute is the only additional attribute required to determine the remaining hierarchical information for the step. Thus, the following attributes can be used to create a covering index for an entire XPath step:

Example 5 (covering index attributes)
name, {branches}, level, type.

These attributes are different from traditional approaches as node labels such as *pre/post* or Dewey decimal [55, 62] are not required to satisfy the hierarchical relationships between nodes. If they were required, the covering index would be at least as large as the original XML document as *pre/post* or Dewey decimal labels are always unique for each node. To exemplify how `name, {branches}, level` and `type` are exploited to satisfy hierarchical relationships, consider the single context node sequence containing node `8` (Figure 4.3). If the step contains the descendant axis, the query process is as follows:

1. Idetify the **descendant-or-self** branches: `LP-4`, `LP-5`, `LP-6`, `LP-7`, `LP-8`, `LB-9`, `LP-10`, `LB-11`, `BLP-12`, `LB-13`.
2. Check the *name, level* and *type* of each node in these branches to determine the sequence of target nodes.

At this point, there are still too many branch instances for this system to be very efficient for most purposes. This provides the motivation for reducing the size of the BranchIndex in Chapter 5. After the size of the BranchIndex is reduced, we will show how the `name, {branches}, level,`

`type` attributes become the basis of a highly optimised covering index for XPath steps.

4.5 Summary

In this chapter, we introduced the constructs that are used in the partitioning process (a *branch* and its sub-types). The creation of primary partitions demonstrated that partitions can be created based on the occurrence of *branching nodes* within an XML document, which avoids preprocessing to identify suitable partitioning factors; an issue we identified for the approach most similar to ours [43] (see Chapter 2). We then showed how an index of primary partitions can result in a large number of *false hits* for the hierarchical XPath axes (as nodes within an individual branch partition may not share a hierarchical association), and discussed their associated performance overhead. To address the issue of false hits, we discussed how partitions can be resized. For this purpose, a new strategy that uses the `RefinePartitions` algorithm was presented, which creates branch partitions using the rule that nodes within a branch *must* be hierarchically related.

Chapter 5

Classification of Partitions

The BranchIndex, resulting from the branch partitioning approach presented in Chapter 4, provides a search space pruning method for XPath steps. However, as we pointed out, this technique has a large index storage cost because the fine granularity of the partitioning process results in a large number of branch instances. Thus, traversals of the BranchIndex have a performance overhead that reduces the gains achieved through search space pruning. In this chapter, we describe a technique which reduces the size of the BranchIndex using a branch classification process. The BranchClassIndex, which is a specialised version of the BranchIndex, maintains all of the benefits for a reduction of the storage and therefore index traversal costs. In addition to *search space pruning*, (as an additional benefit of the branch classification process), we will introduce our *node bypassing* strategy for XPath steps.

This chapter is structured as follows: in §5.1, the branch classification process is introduced; §5.2 describes how the BranchClassIndex is exploited to optimise XPath queries; in §5.3, we describe how the classification process can be extended to achieve further query performance gains; finally in §5.4, we conclude by demonstrating that the integrity of *result sets* is maintained after the classification process.

5.1 Branch Classification

The final phase in constructing the index is to reduce its size while maintaining the same degree of search space pruning - this is process p4 in the system architecture (Classification Process). To achieve this, branch instances are classified into *branch classes* which are used to construct the final index (the BranchClassIndex). In the BranchClassIndex, a single branch class represents many branch instances.

Before presenting the classification process, we will discuss how the properties of the BranchIndex migrate to its specialised version (the BranchClassIndex). Recall from Chapter 4 that based on the special properties of the branch partitioning process and given a sequence of context nodes, the hierarchical XPath axes can be satisfied using the following attributes of a node:

Example 6 (original covering index attributes)

name, {branches}*,* level*,* type*.*

In the same chapter, we also demonstrated how these properties can be exploited to create a covering index for steps in an XPath expression that contains a hierarchical XPath axis. The problem that was noted however, is that there will usually be too many branch instances in an XML document to realise significant performance gains. In contrast, *now* based on the classification of branch instances, an optimised covering index for hierarchical axes can be achieved using the following attributes of a node:

Example 7 (new covering index attributes)

name, {branch classes}*,* level*,* type*.*

We refer to the index that is based on these attributes as the `NCLT` (Name, Class, Level, Type) index. In the `NCLT` index, a single *branch class proxy* represents many branch instances. Therefore, we will show how large numbers

of nodes can be bypassed during the query process. The overall optimisation strategy can now be viewed in two phases:

1. For hierarchical XPath steps that must access the *base data*, the search space is pruned by exploiting the branch information that is implicit in the branch classes that are indexed.

2. For instances in which the `NCLT` covering index can be exploited to optimise an XPath step, large numbers of nodes can be bypassed because a single branch class proxy can be evaluated in place of a large number of branch instances. This chapter will show that, when the `NCLT` index is used, *search space pruning* and *node bypassing* are achieved simultaneously.

Definition 7 ***(Generic Ancestor)***
A generic ancestor is a node that is an ancestor of all nodes in an XML document with the exception of the root and other generic ancestor nodes.

For example, in Figure 5.1 (below), node 1 is a generic ancestor because it is an ancestor of all nodes except the root node. As will be shown shortly, the root node, and generic ancestor nodes, can be ignored during the branch classification. Therefore nodes can be indexed earlier (removed from *main memory*) making the classification process more efficient.

5.1.1 Branch Classification

The branch classification process will place every branch instance in a single branch class. To achieve this, each XML document is treated as a *set* of disjoint *sub-documents* where each sub-document is rooted at the first level that contains a non-generic ancestor. In Figure 5.1, the root node resides at level 0. The only generic ancestor in this example resides at level 1. Thus, sub-documents 1-4 are rooted at level 2.

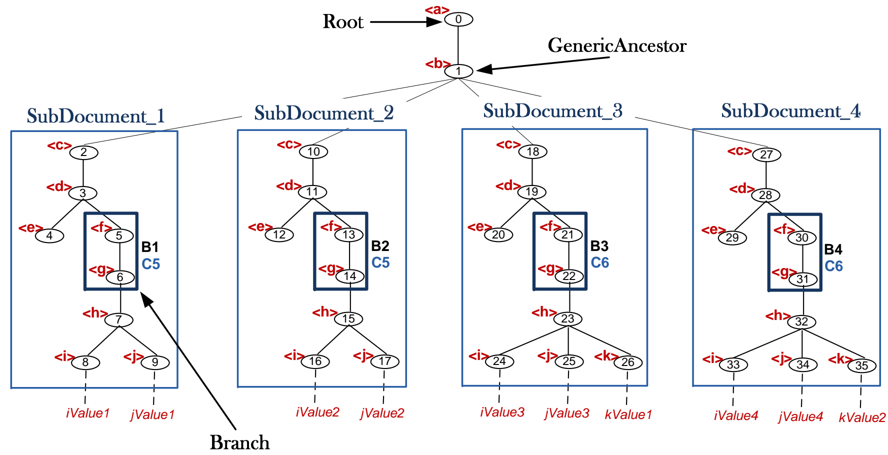


Figure 5.1: XML Tree Showing Branch Classifications

The classification process we are about to present is in some ways similar to the concept of forward and backward bisimulation [1, 35]. The first major difference is that we are classifying branch instances rather than node instances. Secondly, paths leading to element and attribute node names are considered in the branch classification process, but those leading to their text values are not. For example, in Figure 5.1, the text values (e.g. *iValue1*, *kValue1*) are not included in the classification process, therefore they do not reside inside a sub-document.

Example 8 (Forward Paths of Node 3)

/c/d/e-{}, /c/d/f-{*/g, /g/h, /g/h/i, /g/h/j*}, /c/d/f/g-{*/h, /h/i, /h/j*}, /c/d/f/g/h-{*/i, /j*}, /c/d/f/g/h/i-{}, /c/d/f/g/h/j-{}

The steps in the branch classification process for each sub-document are as follows:

Step 1: Calculate Forward Path Identifiers. Traverse each element node in the sub-document and assign to it, a forward path identifier. We calculate this identifier for each element node using the dash separated pair [*root path*-{*ordered outgoing paths*}]. For instance, the forward path for node

3 in Figure 5.1 is shown in Example 8.

Recall from Chapter 4 (XML Document Partitioning) that the partitioning process exploits a stack structure to process element nodes. In short, when an *opening tag* is encountered the new element is pushed onto an *element stack*. When a *closing tag* is encountered, the element on the top of the element stack is *popped* (removed from the stack) because the end of the node has been reached. Using the same stack based approach, the steps in calculating the forward bisimulation are as follows:

1. When an *opening tag* is encountered, add the *root path* (the path containing the names of the nodes from the root of the sub-document to this node) of this new node to the *set* of *forward paths* of each node currently on the element stack.

2. When a *closing tag* is encountered:

   - The top element is popped from the stack, and its [*root path*-{*outgoing paths*}] pair (described above) is added to the *set* of *forward paths* that each node on the element stack maintains.

   - Assign a unique forward identifier to the forward paths of the node just popped from the stack, or, get the previously assigned identifier if the same path was encountered before.

This process has worst case O(N*M) complexity, where N is the number of nodes in the sub-document and M is the maximum depth of the tree structure (the maximum possible number of nodes on the element stack). Theoretically, M can be large, but it is common for the maximum depth of an XML document to be less than 10 levels, for example DBLP [21] (6 levels), Protein Sequence Database [58] (7 levels). In fact, the XPathMark benchmark [67] for stress testing the performance of XPath queries against synthetic XML data uses just 13 levels.

Step 2: Calculate Backward Path Identifiers. Traverse each element node in the sub-document one more time and assign to it, a backward path identifier. A node's backward path identifier is calculated using its *root path'*, where the *root path'* is the path of containing the forward path identifier of each node (calculated in the previous step) from the root node, within the sub-document, to this node.
As with forward path identification, this process has worst case O(N*M) complexity because the element stack is traversed for each node in the sub-document to find the *root path'* of the node.

Step 3: Classify Branch Instances. For each branch, its branch class identifier is calculated as backward path identifier of each element node in the branch, in document order.

5.1.2 Typical Build Times and Storage Costs.

The build times for the four XML datasets evaluated in this dissertation are shown in Table 5.1 - these datasets are formally introduced in Chapter 7. The build times for the same datasets using SQL Server 2008 and MonetDB/XQuery are also shown for comparison purposes. MonetDB/XQuery took the least amount of time across all datasets, followed by the BranchClassIndex and SQL Server, respectively. SQL Server has the least efficient indexing process across all of the XML datasets. Also, for SQL Server, a build time is not shown for the Computer Science Bibliography (DBLP) because it had difficulty processing and XML document with an associated DTD (Document Type Definition).

Dataset	Size	BCI	SQLS	MonetDB	Unit
DBLP	676 MB	4.70	——	1.56	Minutes
XMark	1.33 GB	9.93	36.73	3.34	Minutes
Protein	683 MB	7.86	37.28	1.69	Minutes
Bikes	2.06 GB	22.55	146.62	18.30	Minutes

Table 5.1: Build Times

The BranchClassIndex took more time than MonetDB/XQuery to build the index; the difference for each dataset in minutes is: 3.14 (DBLP), 6.59 (XMark), 6.17 (Protein), 4.25 (Bikes). In other words, for the four datasets respectively, MonetDB/XQuery is 100.30%, 99.32%, 129.21%, 20.8% more efficient at building the index. However, MonetDB requires 'modifications to the internals of the underlying RDBMS kernel', whereas the BranchClassIndex can be deployed in an 'Off-the-Shelf' relational database [31].

Dataset	XML Doc.	BaseData	BCI
DBLP	676 MB	951 MB	269.40 KB
XMark	1.33 GB	1.22 GB	117.23 MB
Protein	683 MB	853 MB	149.76 MB
Bikes	2.06 GB	2.56 GB	80.04 KB

Table 5.2: BranchCLassIndex Storage Costs

The layout of the BranchClassIndex is detailed in Chapter 6. Table 5.2 shows the storage cost (size on disk) of the BranchClassIndex. The BaseData is the representation of the original XML dataset within Oracle 11g relational database and BCI is the storage costs of the BranchClassIndex. For each XML dataset, the BaseData is larger than the original XML document. This is because it contains the encoded (pre/post) structure of the XML document and indexing attributes associated with the BranchClassIndex, such as the branch class identifier of each node, are stored in the BaseData. Crucially however, for each dataset the storage costs of BranchClassIndex is smaller than the BaseData, for example, 269.40 kilobytes (DBLP), 117.23 megabytes (XMark).

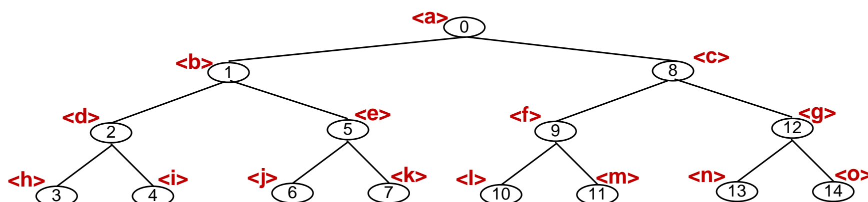


Figure 5.2: Full Binary Tree with Distinct Path for each Node

5.1.3 Worst Case Storage Costs for BranchClassIndex

Recall, from Chapter 4, that the BranchIndex contains the *ancestor/descendant* and *parent/child* relationships between branch instances. Therefore, because the BranchClassIndex is a compact version of the BranchIndex it must contain the *ancestor/descendant* and *parent/child* relationships between branch classes.

In the worst case, there will be as many branch instances as there are nodes in the XML document. For instance, a *full binary tree* (sometimes called a *proper binary tree*) is a tree in which each non-leaf node has exactly two child nodes. According to our definition of branch instances, branching nodes always reside in a single node partition. Thus, if there are no attribute nodes in the XML tree (because they reside in the same branch as their parent element) and it is a full binary tree, each branch instance will contain exactly one node (a single branching node or a single leaf node). Furthermore, if each node in this XML tree has a distinct root path (that is, it does not share its root path with any other node in the tree), there will be as many branch classes are there are nodes in the XML document - such an XML tree is shown in Figure 5.2.

To index the relationships between branch classes in a relational database, we adopted the approach similar to that proposed by XParent [34] (see Chapter 6 for details). In XParent, the relationships between nodes - or the transitive closures of nodes - are stored explicitly in an `Ancestor` table. This ancestor table allows XPath steps to be answered using equi-joins rather than θ-joins (nonequi-joins). Thus, if there are as many branch classes as there are nodes in the XML document, the performance of the BranchClassIndex reduces to that of XParent.

As our approach in its worst case can, like XParent, use equi-joins rather than θ-joins, *hash-join* algorithms are an option of the query optimiser. A Hash join has O(N+M) complexity. However, the inner row of a hash-join

must be loaded into memory. Thus, in our worst case scenario, the query optimiser may choose nested-loops, which has a worst case complexity of O(N*M).

5.2 Exploiting Branch Classification to Optimise XPath Queries

In Chapter 4, we described how branch instances are exploited to prune the search space during query processing. In particular, a description was provided of how the branch instances (that prune the search space) are identified for each XPath step based on its *axis* and *sequence of context nodes*. We now show how this process is optimised using the BranchClassIndex. For this purpose, we introduce the concept of a ClassChain. This is followed by a description of how ClassChains are exploited to improve the performance of XPath steps. The section is completed with a worked example which illustrates how the query process associated with the BranchClassIndex differs from that used by other approaches.

Definition 8 ***(ClassChain)***
For any branch class (which we will call the self-class), a ClassChain contains the self-class and its hierarchically related classes in root-to-leaf order.

The ClassChain is sub-divided into the following three components:

1. `Ancestor-Component` - contains the classes that are ancestors of the self-class. Contained within this component is the `Parent-Component` which contains the classes that are the self-class' parents.

2. `Class-Component` - contains the self-class only.

3. `Descendant-Component` - contains the classes that are descendants of the self-class. Contained within this component is the `Child-Component` which contains the classes that are children of the self-class.

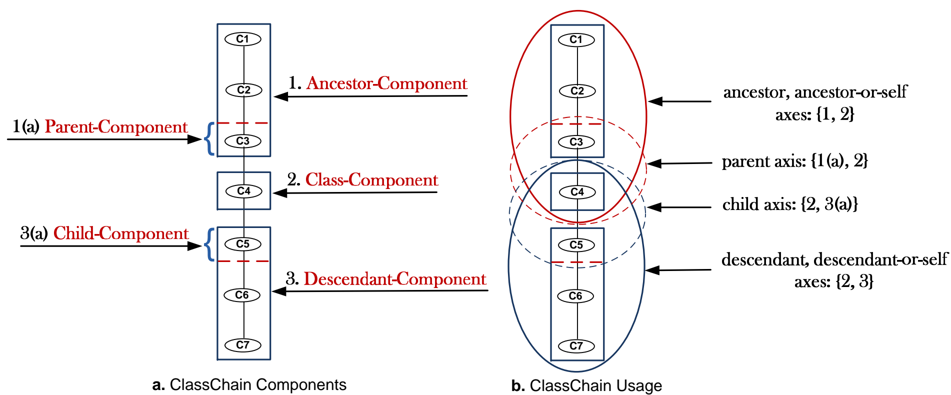


Figure 5.3: ClassChain Components and Usage

Figure 5.3a shows the ClassChain components associated with self-class `C4`. Figure 5.3b illustrates how these components are exploited to evaluate the hierarchical XPath axes:

- A combination of the `Class-Component` and the `Ancestor-Component` is used for the ancestor and ancestor-or-self XPath axes.
- Components `Parent-Component` and `Class-Component` are used for the parent axis.
- Components `Class-Component` and `Child-Component` are used for the child axis.
- Components `Class-Component` and `Descendant-Component` are used for the descendant axis.

The `Parent-Component` and the `Child-Component` are *optional* indexing constructs. They provide additional performance across the *parent* and *child* axes because they eliminate a larger number of branch classes from the query process.

5.2.1 Modelling the Indexing Constructs

In this section, we describe how the optimisation constructs are modelled. The Class Diagram in Figure 5.4 shows that each Node has a NodeLabel. A NodeLabel captures a node's relationship to other nodes. In this instance, we chose preorder and postorder labels (i.e. `pre/post` encoding). However, as our approach is encoding scheme independent, all XML encoding schemes are permitted in the NodeLabel. For example, `ORDPATH` encoding [55] is more appropriate for systems that require frequent updates, whereas pre/post encoding is best suited to *read only* XML databases. A Node has the additional attributes: *name* (the node's name), *value* (text values), *type* (element/attribute), and *level* (distance from the document node), which capture its properties within the document.

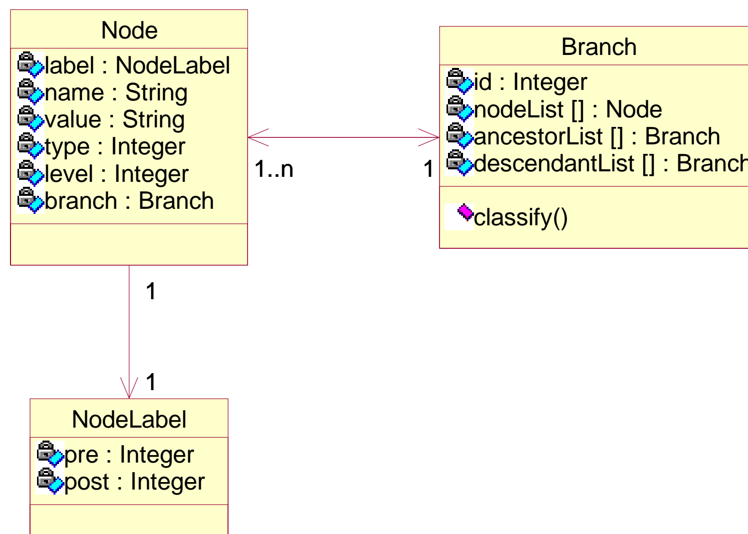


Figure 5.4: Optimisation Constructs

In the storage model, one or more nodes *belong to* a single branch instance and a branch instance *has* one or more nodes (its nodeList[]). Each branch instance has a `classify()` function that places the branch in the relevant branch class. Thus, *id* in Branch will be the branch class identifier of the branch after it has been classified. In the ClassChain (Figure 5.3), this *id* is

the Class-Component. In the Branch class the two attributes *ancestorList[]* and *descendantList[]* contain the *set* of ancestor branch instances and descendant branch instances respectively (before the `classify()` function has been called). After classification, they contain the ancestor branch classes (Ancestor-Component in the ClassChain) and descendant branch classes (Descendant-Component in the ClassChain) respectively.

5.2.2 Worked Example

We will now provide a worked example to illustrate how optimisation is achieved for hierarchical XPath steps using the BranchClassIndex. Figure 5.5 depicts a small segment taken from the bicycle rental repository. Each node in the XML tree is identified by its *preorder* label. The branch class for each node is also shown; for example, nodes 2, 3, 4, 15, 16, and 17 are in branch class `C5`.

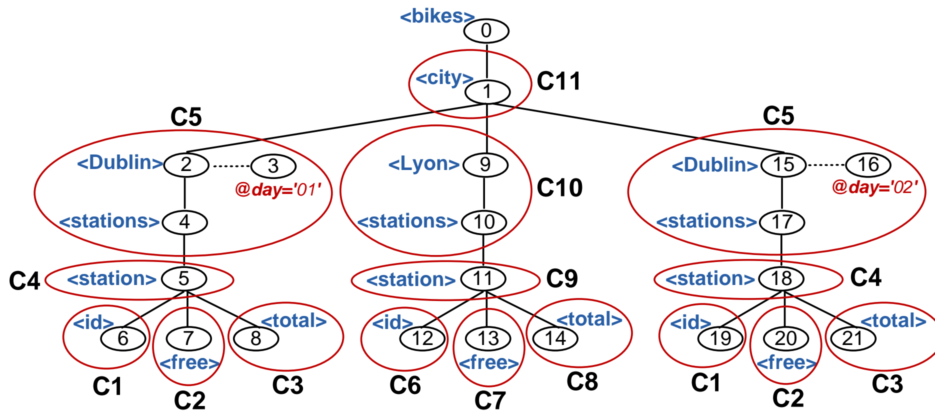


Figure 5.5: Bicycle Rental Repository Subset

The following example contrasts our query process to that of similar approaches [29, 31]. The goal is to illustrate the two key concepts of the BranchClassIndex: (1) *search space pruning* and (2) *node bypassing*.

Example 9 *Return all bicycle stations in Dublin.*

//Dublin/stations/station

pre	post	name	class	level	type
2	6	Dublin	5	2	1
4	5	stations	5	3	1
5	4	station	4	4	1
15	19	Dublin	5	2	1
17	18	stations	5	3	1
18	17	station	4	4	1

Table 5.3: Extract from the Node Repository (Base Data)

The XPath query in Example 9 contains three steps, each of which locates nodes within the XML tree as follows:

//Dublin. The first step contains the *descendant-or-self* axis. As it is the first step in the expression, the nodes that are descendant (or self) of the document node (*node 0*) must be located. The NodeTest specifies that those nodes must be *element* nodes that have the name *Dublin*.

- **Traditional Approach.** Identify a sequence of element nodes called *Dublin*, which is the sequence of nodes: (2, 15) - see Table 5.3.
- **BranchClassIndex Approach.** The proxy for class `C5` in the `NCLT` index is shown in Table 5.4. Thus, in an instance of *node bypassing*, the `NCLT` index is exploited to identify a *single* proxy: `p1`, which represents both node 2 and node 15. Proxies `p1-p3` are placed in Table 5.4 for illustration purposes only; these identifiers do not appear in the index.

Essentially, in the first XPath step, the traditional approach identifies two nodes in the node repository, whereas the BranchClassIndex approach identifies a single proxy node in the much smaller `NCLT` index that represents both nodes.

/stations. Identify the element nodes called *stations* that are children of the nodes identified at the first step.

id	name	class	level	type
p1	Dublin	5	2	1
p2	stations	5	3	1
p3	day	5	3	2

Table 5.4: Extract from the NCLT Covering Index

- **Traditional Approach.** The context node sequence is (2, 15). Therefore, using *pre/post* labels (see Table 5.3), node 2 is evaluated to find all nodes in the node repository that have a preorder identifier greater than that of node 2 and a postorder label less than that of node 2. Then, make sure it is a child node using the node's *level* attribute - the node is a child if it resides at one level greater than that of the context node. The process is then repeated for node 15. The target node sequence (4, 17) is identified.

- **BranchClassIndex Approach.** This is the first step in which the ClassChain is utilised. As the step contains the child axis, ClassChain components: 2 (`Class-Component`) and 3(a) (`Child-Component`) are selected (recall ClassChain component usage Figure 5.3); these components contain branch classes 4 and 5 (`C4` and `C5` in Figure 5.5). It must now be determined if it is necessary to access the base data or to use the `NCLT` covering index. The query process uses these ClassChain components to *prune the search space* in either case because all classes that are not in the class chain are avoided. The index selection process is detailed in Chapter 6; in this particular instance, it will choose the `NCLT` index.

  The input to this step was the sequence of proxy identifiers: (`p1`). Thus, using the attributes of proxy `p1` (Table 5.4), in an instance of *node bypassing*, the query process will query the `NCLT` index and find the nodes called (*Name*) *stations*; that are in (*Class*) class 4 or 5; that have a level (*Level*) greater than that of `Dublin` (i.e. must be greater

than 2); and finally (*Type*), it must be of type 1 (an element node). The output is therefore the sequence (`p2`), which represents both node 4 and node 17.

The traditional approach processes each node in the context node sequence and identifies all target nodes for every step in the expression. In contrast, for the BranchClassIndex, the query process receives a sequence of proxy representatives and (1) prunes the search space using the ClassChain, and (2) bypasses nodes by identifying a sequence of target proxy representatives, rather than a sequence of target nodes. In this instance, the sequence (`p2`) represents nodes 4 and 17 in the base data. In practice, a proxy will often (see Chapter 7 for details) represent thousands and even millions of nodes in large XML datasets.

/station. Identify the element nodes called: *station* that are children of the nodes identified at the previous step.

- **Traditional Approach.** The context node sequence is (4, 7). Again, each node is evaluated in turn and using the context node's attributes (e.g. *preorder, postorder, level*); target nodes (5, 18) are identified. As this is the rightmost step in the expression, nodes 5 and 18 are returned.

- **BranchClassIndex Approach.** The sequence of context proxies contains `p2`. As it is the rightmost step in the expression, the `NCLT` index cannot be used as the actual nodes must be retrieved and returned as the result set. In other words, node bypassing is not possible for the rightmost step in any XPath expression (see *index selection* - Chapter 6 - for more details).

  However, the ClassChain can be exploited to prune the search space. As the axis is *child*, ClassChain components associated with the classes

in the context node sequence (p2) must be identified. Thus, the components that are associated with class 5 (C5 - Figure 5.5) are selected: 2 (Class-Component) and 3(a) (Child-Component), which contain classes 4 and 5 (C4 and C5 Figure 5.5). It remains to find all nodes in the base data that have the name *station*; are in class 4 or 5; have a level one greater than 3; and a type of 1 (an element node). The node set {5, 18} is therefore returned.

5.2.3 Worked Example Summary

The result set for the XPath expression in Example 9 is {5, 18} and both approaches returned the same result. However, the traditional approach visited nodes (2, 4, 5, 15, 17, 18) using inefficient *pre/post/level* node comparisons. In contrast, the BranchClassIndex approach evaluated proxies (p1, p2) - which represented nodes (2, 4, 15, 17) - and the search space was pruned using the class chain in each case. Finally, nodes 5 and 18 are the result nodes for the query, thus the NCLT index could not be used. When locating these nodes in the base node repository, the BranchClassIndex pruned the search space by evaluating *only* those nodes in classes C4 and C5 - all other branch classes were avoided.

5.3 Extending the Branch Classification Process

In this section, we show how some XPath steps can be optimised by extending the classification process to include text values (recall that they were omitted from the classification process to minimise the number of branch classes). However, we only allow certain text values to be classified (those that do not increase the size of the index significantly). Using the same terminology as in [47], these text values can be categorised as *high selectivity* and *low selectivity* text values.

If a text value has *high selectivity*, it will occur a small number of times in the base data. Thus, in spite of the fact that the base data must be evaluated (when an XPath query is predicated on a text value) text values that have high selectivity lead to fewer node comparisons then if they have low selectivity. In contrast, when an XPath expression is predicated on a text value that has low selectivity, there will be a large number of inefficient node comparisons. This is because text values that have low selectivity occur many times in the base data.

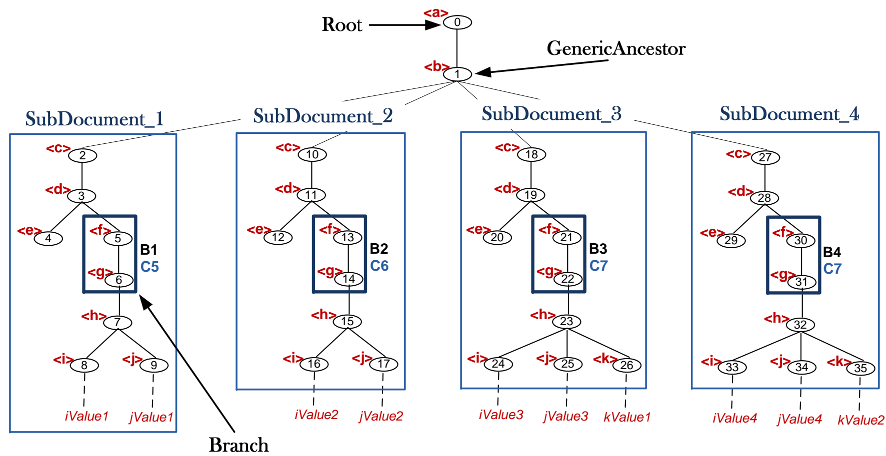


Figure 5.6: XML Tree Showing Branch Classifications

To optimise the BranchClassIndex for queries that evaluate text nodes that have low selectivity, the paths to certain text values are included in the classification process. For example, in Figure 5.6, notice the how text values *iValue1* and *jValue1* are within SubDocument_1, which means that the paths to these text values are included when calculating the forward paths of element nodes in that sub-document. This has the effect of placing branches `B1` and `B2` in different branch classes, whereas before (see Figure 5.1) they where placed in the same branch class.

As we demonstrate empirically in Chapter 7, many text values that have low selectivity can be classified while introducing only a small number of additional branch classes. This has the effect of optimising XPath expres-

sions text values that have low selectivity for only a small increase to the index size. Our goal is to create a situation where the only time the base data is evaluated will be to evaluate text values that have high selectivity. Therefore, improving the generic performance of the index and providing a lever to control the space/time (index size/query performance) trade off.

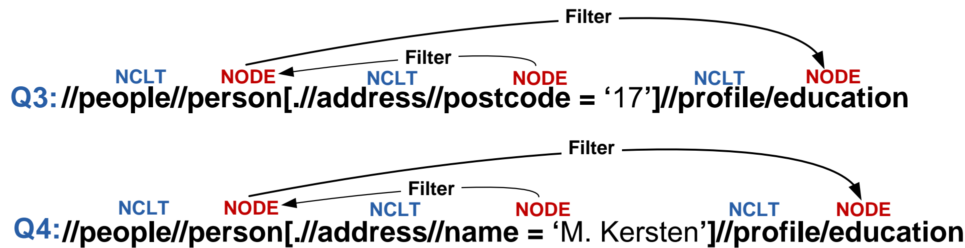


Figure 5.7: Illustrating the Effect of Low Cardinality Text Values

To illustrate why optimisation is only achieved through the classification of text values that have *low* selectivity, consider the two queries in Figure 5.7. Query `Q3` must access the base data at step two because the text value '17' must be evaluated. In a real world scenario, there can be many thousands of people living in the same postcode (e.g. at postcode 17). Thus, there can be thousands of node comparisons as each *person* residing at postcode 17 will be evaluated. In contrast, it is unlikely (see `Q4`) that there will be thousands of people with identical names, i.e. *M. Kersten*, living in the same postcode. In this situation, even though the base data must be accessed in step two of `Q4`, there may be few node comparisons required (just one for each person with the name *M. Kersten*).

5.3.1 Identifying Text Values that have Low Selectivity

In order to classify text values that have low selectivity, the classification process must be able to detect them. We propose two ways in which information relating to these text values can be retrieved in advance of the classification process:

1. **Domain Knowledge.** Knowledge of the data is known in advance. For example, if one knows that the dataset contains an element called *gender*, the classification process can be told to include text values that have low selectivity such as *male* and *female*.

2. **Text Value Identification Algorithms.** These algorithms will identify text values that are suitable for classification.

The first approach requires no further details, because prior knowledge of data will already be available by some other means. Therefore, we will focus on the second (identification algorithms) approach, which is applicable to all XML datasets.

Q5: //people//person[.//address//postcode = '17']//profile/education

Q6: //people//person[.//address//postcode > '17']//profile/education

Figure 5.8: Examining Text Value Operations

There are a couple of considerations to be aware of when classifying text values that have low selectivity. For example query `Q5` in Figure 5.8, specifies that all postcodes with a text value *equal to* 17 are evaluated. Conversely, in `Q6` the *range* operator ('>') requires that all postcodes *greater than* 17 be evaluated. In order to perform range evaluations across text values associated with a given node, *all* of the text values for that node must be classified (not just individual text values). In other words, it is not sufficient to classify the text value '*17*' for the element node *postcode*; all text values associated with *postcode* must be classified.

Classifying all of the text values associated with a given node can lead to a large increase in the number of branch classes and therefore, the size of the overall index. In contrast, for evaluations based on the equality operator ('='), it is sufficient to classify individual text values associated with a given node (all of its text values do not have to be classified). Thus, the process

for identifying text values that have low selectivity is broken into two parts:

1. **Range Based Classification.** The classification of all text values associated with a node if there is a small number of distinct text values for a large number of nodes. For example, there can be a large number of nodes called *gender* with only two distinct values *male* and *female*. In this instance, all of the values associated with nodes called *gender* are classified.

2. **Equality Based Classification.** Classification of individual text values that have a cardinality greater than some threshold (thresholds are discussed below). Only the equality operator ('=') will be permitted in XPath steps that evaluate these text values (range based text value evaluations must still access the base node repository).

5.3.2 The Text Value Identification Algorithms

Firstly, the algorithm for identifying text values that are candidates for range based classification is provided (`IdentifyRangeBasedCandidates`). Secondly, the `IdentifyEqualityBasedCandidates` algorithm shows how individual text values that are suitable for classification are identified. A single pass of the XML document is required in advance of the classification process to identify the input values for the algorithms. The performance of both algorithms and the pass of the XML document are discussed at the end of this section.

The Range Based Classification Algorithm

As discussed, to support range based evaluations across text values, all of the text values associated with a node of a given *name* and *type* (element/attribute) must be classified. The `IdentifyRangeBasedCandidates` process is designed to identify nodes that can be classified while incurring

only a small increase to the number of branch classes.

Algorithm 2 IdentifyRangeBasedCandidates

```
1: NodeMap := (NodeName ↦ TextValues);
2: NodeCardinalityMap := (NodeName ↦ Cardinality);
3: for NodeName in NodeMap do
4:    CardinalityOfTextValues := NodeMap.get(NodeName).size();
5:    CardinalityOfTotalNodes := NodeCardinalityMap.get(NodeName);
6:    if CardinalityOfTotalNodes ≥ (CardinalityOfTextValues * N) then
7:       classify all text values associated with NodeName;
8:    end if
9: end for
```

As stated earlier, the text value identification processes require one additional pass of the XML document in advance of the classification process. In `IdentifyRangeBasedCandidates`, *lines 1* and *2* are populated during this pass of the XML document. NodeMap (*line 1*) is a *map* data structure containing each NodeName mapped to its *set* of TextValues. For example, if the XML document contains a node called *gender*, after document pass the NodeMap could contain the entry: *gender* $\mapsto$ {*male, female*}. Similarly, the NodeCardinalityMap contains each NodeName mapped to its Cardinality. Thus, if there were 23 males and 48 females in the XML document, the NodeCardinalityMap would contain: *gender* $\mapsto$ *71*.

For range based classification, we are only interested in nodes that have a small number of distinct text values (such as *male*, *female*) for a large number of nodes (such as *gender*) because their text values will have low selectivity. Thus, to optimise the document pass, we quickly eliminate nodes that have too many distinct values. For example, if an upper threshold of 100 distinct values is used, at the end of the document pass, the NodeMap and the NodeCardinalityMap will only contain nodes that have fewer than (or equal to) 100 distinct values. We discuss suitable thresholds in Chapter 7.

To determine if there is a small number of distinct text values for a large

number of nodes, for each node of a given name (NodeName) in the XML document (*line 3*), the process identifies its total number of distinct text values (CardinalityOfTextValues) (*line 4*) and the total number of nodes with that name (CardinalityTotalNodes - *line 5*).

If the total number of nodes (CardinalityOfTotalNodes) that have a given name and type is small, there is no point in classifying its text values. To see why, consider the node *gender* again. If there are only `100` nodes called *gender* in the document, there is no point in classifying the text values associated with gender, even if there are only two values (*male* and *female*). The reason for this (as discussed earlier) is that even if the base data must be accessed to evaluate nodes called *gender*, it will be efficient because there will be a maximum of `100` node comparisons.

To determine if the selectivity of the text values are low enough, a threshold `T` must be known. `T` is time it takes to perform a join between nodes in the base data, when an XPath query is predicated on a text value. Say `CN` (Context Nodes) * `TN` (Target Nodes) is the total number of node joins that can be performed when `T` is equal to 1000ms (one second). If we classified all text values that occur more than `N` times in the base data, all of the text values that are not indexed will occur less than `N` times in the base data. Therefore, we would expect that any time the base data must be evaluated (because of a text value in an XPath step) the join operation would be performed in less than one second (we identify suitable `T` values empirically in Chapter 7).

For example, if there are `10,000` nodes called *gender* which have two distinct values: *male* and *female*, *line 6* in IdentifyEqualityBasedCandidates would be as follows: **if** `10,000` $\geq$ (`2` * `T`). If for example, `T` is set to 3000, at *line 7* the text values associated with nodes called *gender* will be classified. In contrast, if CardinalityOfTotalNodes was 5,000, text values associated with gender would not be classified.

The Equality Based Classification Algorithm

A node can have associated text values that have low selectivity while also having text values that have high selectivity. In this instance, range based classification is not suitable because classifying all of the node's text values (including those with high selectivity) could lead to a large increase to the index size. However, it is easy to determine if the XPath step is performing an equality evaluation or a range based evaluation across a text value. Thus, we can extend the index further (and avoid the base data for a greater number of XPath steps) by classifying text values that are suitable for equality evaluations only.

Algorithm 3 IdentifyEqualityBasedCandidates

1: NodeMap := (NodeName $\mapsto$ (TextValue $\mapsto$ Cardinality));
2: **for** NodeName in NodeMap **do**
3:    **for** each TextValue associated with NodeName **do**
4:       TextValueCardinality := NodeName.get(TextValue);
5:       **if** TextValueCardinality $\geq$ `N` **then**
6:          classify Text value for equijoin on NodeName;
7:       **end if**
8:    **end for**
9: **end for**

During the pass of the XML document, a NodeMap is populated which contains each NodeName mapped to another map that contains its text values mapped their cardinality (*line 1*). For each node name in the NodeMap (*line 2*) and for each of its text values (*line 3*) the cardinality of the text values (TextValueCardinality) is identified (*line 4*). If TextValueCardinality is greater than some threshold `N` (*line 5*), it is classified for equality evaluations in XPath expressions (*line 6*). In other words, if the number of times the text value occurs in the XML document is greater than `N` (again this `N` value is determined empirically in Chapter 7), it has low selectivity.

Efficiency Evaluation

We will now discuss the efficiency of the `IdentifyRangeBasedCandidates` and `IdentifyEqalityBasedCandidates` processes.

Machine	Platform	CPU	RAM
Dell	Windows 7 Pro (32bit)	3.00GHz Intel Core Duo	4GB

Table 5.5: System Specification

Dataset	Size	Pass Time	IRBC	IEBC
Bicycle Rental	2.06 GB	2.7 min	1 ms	11 ms
DBLP	676 MB	0.7 min	0 ms	18 ms
XMark	1.33 GB	0.8 min	1 ms	20 ms
Protein	683 MB	0.6 min	0 ms	28 ms

Table 5.6: Efficiency of the Text Value Identification Algorithms

The processes were executed on a system with the specification shown in Table 5.5 and they were implemented in the Java programming language. Table 5.6 shows four datasets, their size, the document pass time in minutes, the `IdentifyRangeBasedCandidates` (IRBC) time in milliseconds, and the `IdentifyEqalityBasedCandidates` (IEBC) time in milliseconds respectively. The largest quantity of time is spent on the document pass, which identifies the input values for both processes. After the document pass is complete, the values in Table 5.6 show that the processes take very little time (less than one second for each dataset).

5.4 Post Classification Integrity

At this point, we have described the branch classification process in full. Additionally, in §5.2, we described how branch classes are used in the query process, that is, how the ClassChain components are utilised at each step in an XPath expression. The objective of this section is to ensure that the integrity of query *result sets* are maintained after branch classification.

The BranchClassIndex works on the basis that `name, {branch classes}, level, type, value` (i.e. the `NCLTV` index) can be used to evaluate hierarchical XPath steps. The `NCLTV` index contains a *branch class proxy* that represents many branch instances. Thus, as the `NCLTV` index is used in place of individual node labels such as *pre/post*, we must ensure that the following IntegrityProposition holds:

IntegrityProposition. If a single branch instance in a branch class contains target node(s), all branch instances in that class will contain equivalent target node(s).

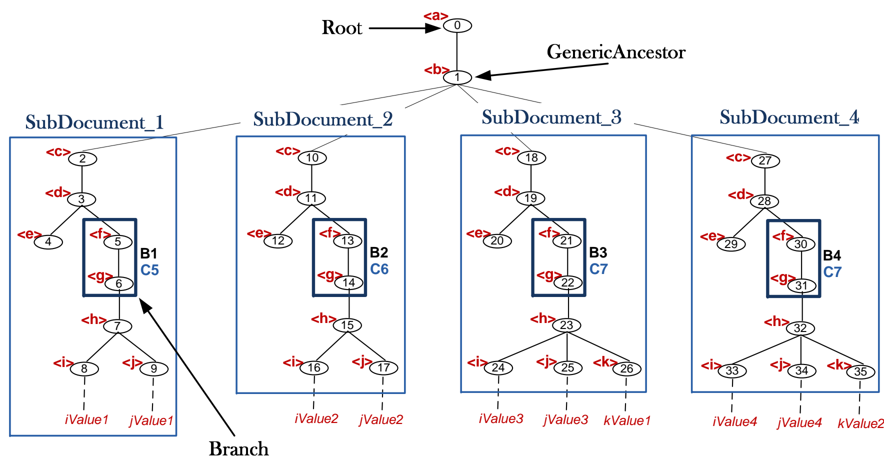


Figure 5.9: XML Tree Showing Branch Classifications

The use of the word *equivalent* in referring to one or more nodes in the following deductions means: a *set* (sorted in document order) of nodes that have the same *name*, *type*, *level* and *value* (*value* only includes the text values that were selected for range or equality based classification).

Example 10 ***(Sample Linear XPath Expression)***

/a//c//h/ancestor::f.

Case 1 (XPath Expressions). Given any linear XPath expression containing one of the six hierarchical XPath axes (such as Example 10), we can deduce that if a single branch instance (`B3` Figure 5.9) contains a target node (node `21`), then all branch instances in the same class (`B4`) will also contain a target node (nodes `30`). The reason for this is that based on the *branch classification*, all branch instances have equivalent forward and backward structure.

Example 11 ***(Sample XPath Twig Query with Text Value)***
/a//d//i[./@key = 'iValue1']/ancestor::f.

Case 2 (Twig Query with Text Value). When a Twig query contains a low selectivity text value that has been selected for classification, the path containing nodes from the root node to the text value itself is included in the classification process. For example, branches `B1` and `B2` are placed in different branches because text values '*iValue1*' and '*jValue1*' were classified. Thus, for Example 11, the result node is node 5; node 13 was not returned because `B1` and `B2` have different forward and backward structure.

5.5 Branch Classification Summary

In this chapter, we described how the BranchIndex can be compacted. To achieve this, we showed how branch instances can be classified with the result that the performance benefits of the BranchIndex can be achieved for a fraction of the storage (and therefore) index traversal costs. This specialised version of the BranchIndex is called the BranchClassIndex. The classification process was then detailed and a description of the query process that exploits the BranchClassIndex for query performance was provided. For this purpose, the concept of a ClassChain was introduced. In particular, we illustrated how the ClassChain was used to provide a *search space pruning*

facility. We then introduced *node bypassing* based on the `NCLT` covering index. Finally, it was shown how the classification process can be extended to include *text values* that have low selectivity (to produce the `NCLTV` index). This leads to overall performance gains because the number of node comparisons that will be required when the base data must be accessed is reduced.

Chapter 6

Query Processing

At this point, we have presented a node partitioning process for XML documents and defined those partitions as *branches*. A description of how the search space can be pruned by indexing branch instances was also provided. In practice however, we showed how may be too many branch instances (in an XML document) to achieve significant optimisation. Therefore, a classification process for branch instances was provided in chapter 5. We demonstrated how the resulting BranchClassIndex, which is a specialised version of the BranchIndex, achieves search space pruning for reduced storage costs. Additionally, it was demonstrated that the properties of branch instances (that are implicit within the indexed branch classes) can be exploited to facilitate the concept of *node bypassing* using the `NCLTV` covering index for XPath steps.

This chapter now describes how these concepts can be exploited to optimise XPath expressions (processes `p11`, `p13` and `p14`). There are two ways of deploying this storage model, native XML indexing (non-relational) or XML-enabled (relational) indexing. The second was chosen because, as we will show, the BranchClassIndex is suitable for deployment in an off-the-shelf relational database, the many advantages of which were discussed in Chapter 2. Relational index deployment is detailed in §6.1. In §6.2, the

XPath-to-SQL transformation is presented. This is followed in §6.3, with a worked example to illustrate the transformation process. Finally, an index selection process is required to determine when the base data must be evaluated and when the covering index can be exploited; this index selection process is presented in §6.4.

6.1 Index Deployment

Figure 6.2 illustrates how the XML document in Figure 6.1 is transformed into four relations. The `Node` relation in Figure 6.2b1 is the base data (construct `i6` in the system architecture). Attributes *pre* and *post* are the preorder and postorder labels of each node respectively. Attribute *name* is the node's name; *type* differentiates between element attribute and document nodes; *level* is the node's distance from the document node; and *value* is the text value associated with the node (or *null*).

The `NCLTV` relation (6.2b2) is populated by selecting distinct *name, class, level, type* and *value* from the `Node` relation (but *value* is included only for low selectivity text values). The *class* attribute in the `Node` and `NCLTV` relations is a node's *branch class identifier*.

The Parent and Child ClassChain components are deployed in the `PC_REL` relation (6.2b3). The `AD_REL` relation (6.2b4) contains the Ancestor and Descendant components. In the `PC_REL` relation, attribute *pc* contains the **p**arent-and-self branch **c**lasses (Parent-Component and Class-Component combination) and attribute *cc* contains the **c**hild-and-self branch **c**lasses (Child-Component and Class-Component combination). In the `AD_REL` relation, the attribute *ac* is the **a**ncestor-and-self branch **c**lasses (Ancestor-Component and Class-Component combination) and *dc* is the **d**escendant-and-self branch **c**lasses (Descendant-Component and Class-Component combination).

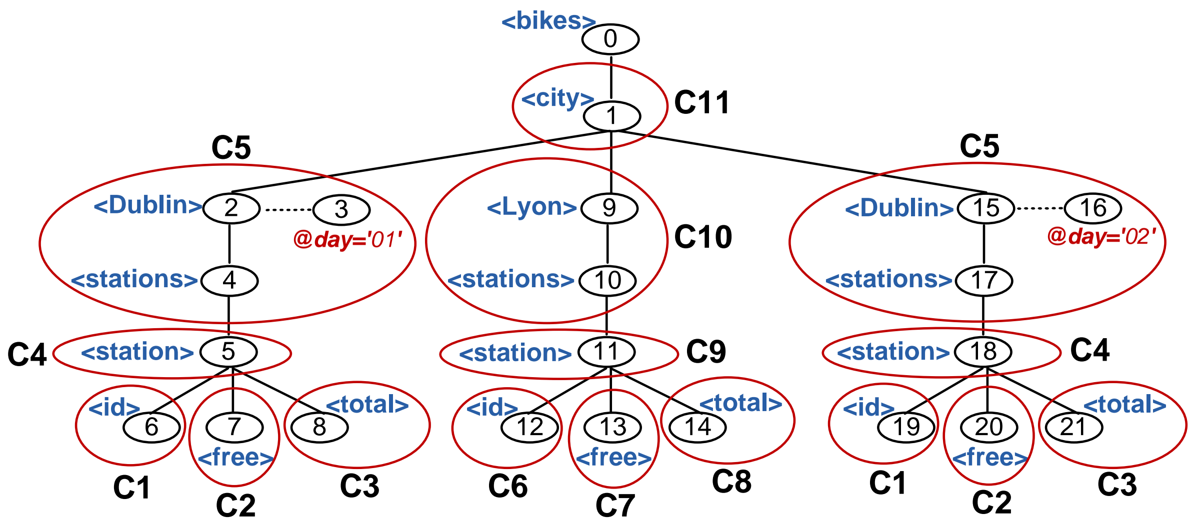
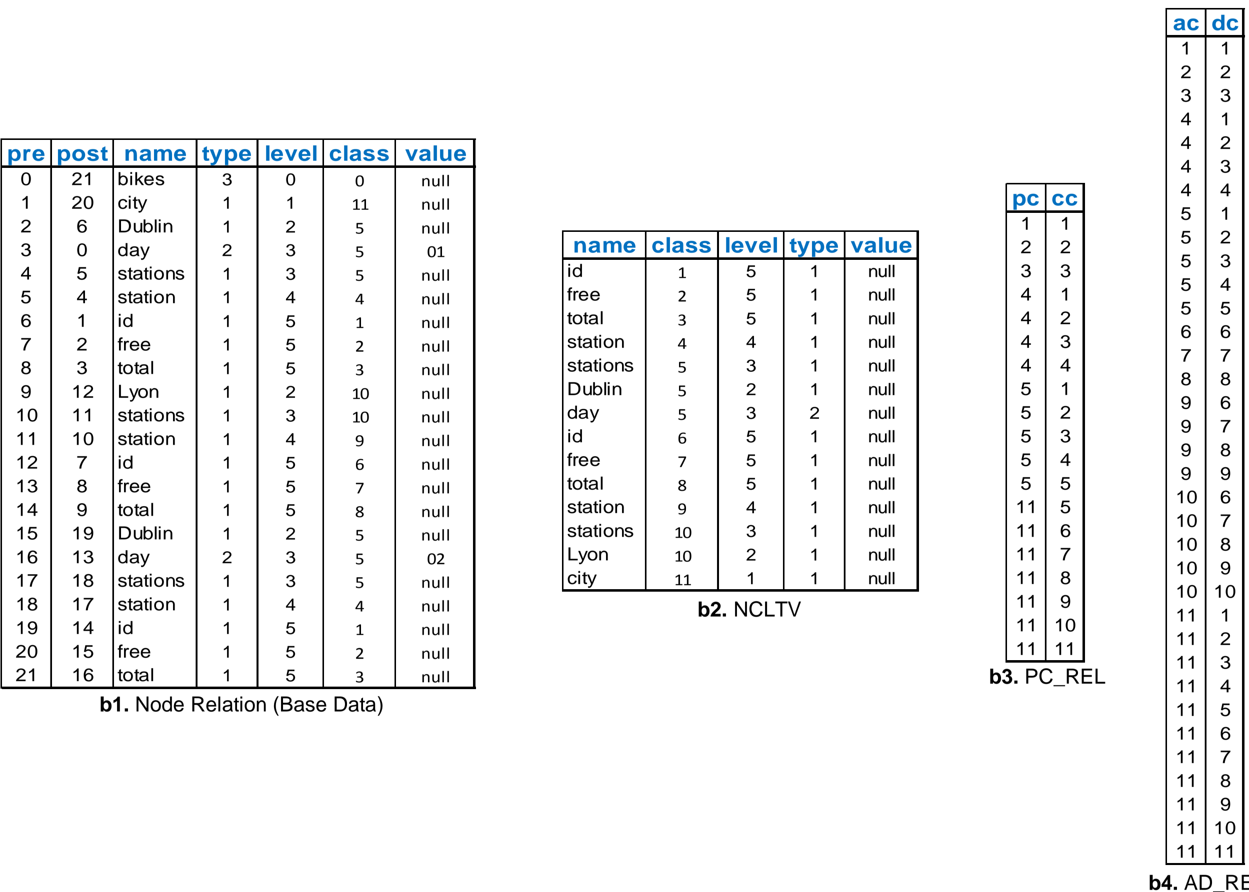


Figure 6.1: XML Snippet from the Bicycle Rental Dataset



pre	post	name	type	level	class	value
0	21	bikes	3	0	0	null
1	20	city	1	1	11	null
2	6	Dublin	1	2	5	null
3	0	day	2	3	5	01
4	5	stations	1	3	5	null
5	4	station	1	4	4	null
6	1	id	1	5	1	null
7	2	free	1	5	2	null
8	3	total	1	5	3	null
9	12	Lyon	1	2	10	null
10	11	stations	1	3	10	null
11	10	station	1	4	9	null
12	7	id	1	5	6	null
13	8	free	1	5	7	null
14	9	total	1	5	8	null
15	19	Dublin	1	2	5	null
16	13	day	2	3	5	02
17	18	stations	1	3	5	null
18	17	station	1	4	4	null
19	14	id	1	5	1	null
20	15	free	1	5	2	null
21	16	total	1	5	3	null

b1. Node Relation (Base Data)

name	class	level	type	value
id	1	5	1	null
free	2	5	1	null
total	3	5	1	null
station	4	4	1	null
stations	5	3	1	null
Dublin	5	2	1	null
day	5	3	2	null
id	6	5	1	null
free	7	5	1	null
total	8	5	1	null
station	9	4	1	null
stations	10	3	1	null
Lyon	10	2	1	null
city	11	1	1	null

b2. NCLTV

pc	cc
1	1
2	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
11	5
11	6
11	7
11	8
11	9
11	10
11	11

b3. PC_REL

ac	dc
1	1
2	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
6	6
7	7
8	8
9	6
9	7
9	8
9	9
10	6
10	7
10	8
10	9
10	10
11	1
11	2
11	3
11	4
11	5
11	6
11	7
11	8
11	9
11	10
11	11

b4. AD_REL

Figure 6.2: Node Relation and the NCLTV Index Relations

6.2 The Transformation Method

In this section, we describe how an XPath expression is transformed into its SQL equivalent. We begin with a desciption of our Transformation Template and then describe the method used to construct the SQL expression using this template to extract the necessary information from the XPath query expression.

The indexing constructs are used to improve the performance of XPath steps that contain one of the six hierarchical axes. Although optimisation specific to other axes in not currently provided, we discuss the future inclusion of these axes (such as *following* and *preceding*) as part of our future work. Thus, the axes shown in Table 6.1 constitute the subset (the most commonly used axes) of the XPath 2.0 language covered here

XPath Axis	**Description**
ancestor	Ancestors of x.
ancestor-or-self	Node$x$ and its ancestors.
descendant	Descendant of x.
descendant-or-self	Node x and its descendants.
parent	The parent of x.
child	The children of x.

Table 6.1: XPath 2.0 Language Coverage

The transformation template (Figure 6.3) represents an SQL expression, which we refer to as the BranchClass expression. It is divided into sub-expressions. These sub-expressions are populated from an expression using a series of transformations rules, which are now described.

Name	**Component**
`C1`	`SELECT`
`C2`	$c_{n-1}.classop$
`C3`	`FROM`
`C4`	[$indexop$ i_x, CLASS c_x]+

Table 6.2: The Generic Expression Components

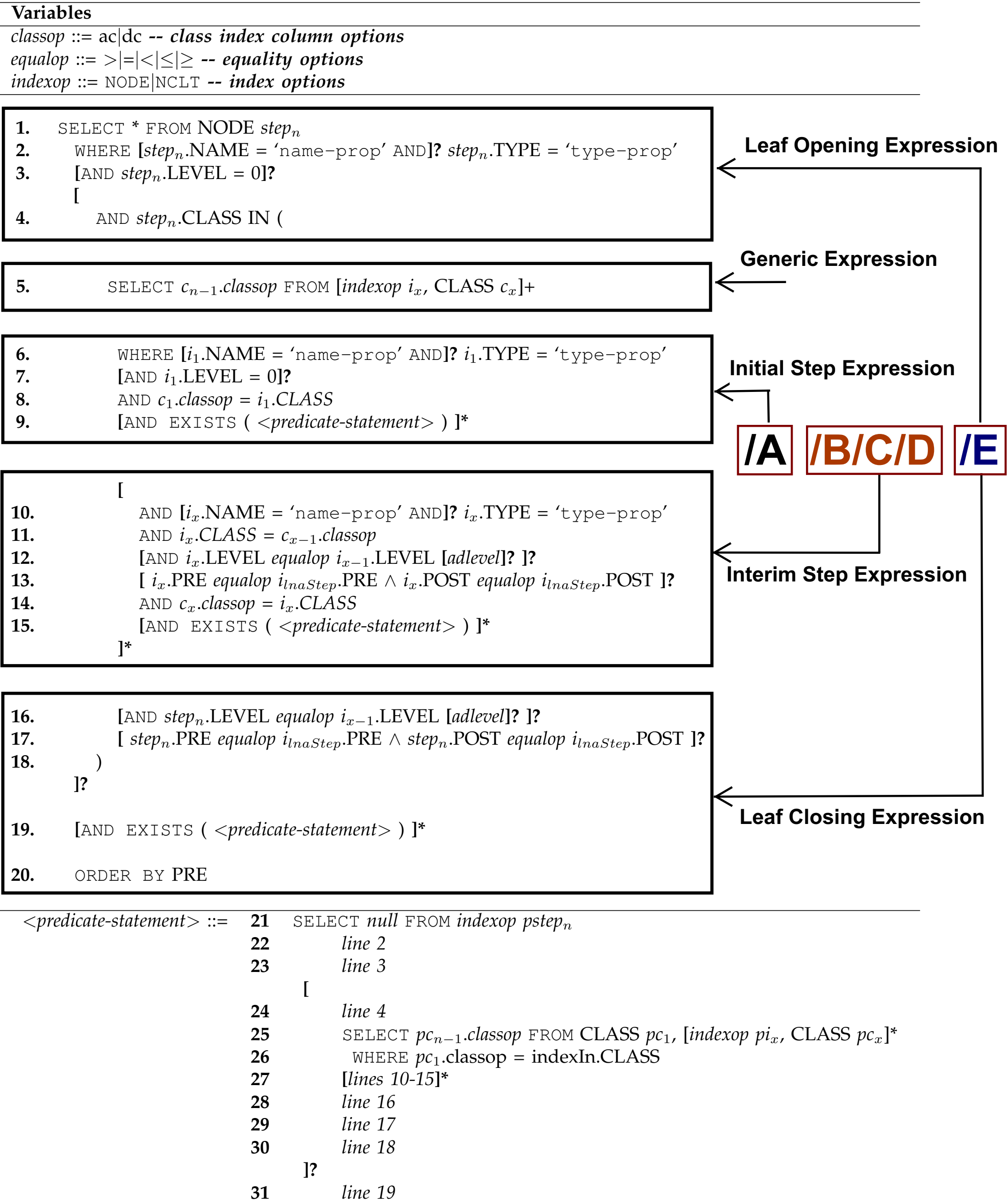


Figure 6.3: The XPath-to-SQL Transformation Template

6.2.1 Generic Expression

Within the transformation template and unlike all other expressions, the generic expression is constructed using properties from all steps in the XPath expression. Its four components are shown in Table 6.2. We will now describe how the generic expression is populated

In the SQL expression, the required tuples are selected from relevant relations using the keywords in components `C1` and `C3`. The tuples that must be returned are specified by `C2` and the relations that contain these tuples are specified in `C4`. The *classop* (class options variable) in `C2` is populated by looking ahead to the last step in the XPath expression and extracting its axis. The `LastAxisProperty` (the axis at the rightmost step) is used for this purpose:

LastAxisProperty. The XPath axis associated with the last step. This enables the branch classes that must be evaluated at the last step to be identified. Using the `LastAxisProperty`, the class options are as shown in Table 6.3.

classop	Selected when...
dc	`LastAxisProperty` is the: *descendant*, *descendant-or-self*, or *child* axis.
ac	`LastAxisProperty` is the: *ancestor*, *ancestor-or-self*, or *parent* axis.

Table 6.3: Populating the Class Option Variable

As shown in Table 6.3, if the `LastAxisProperty` is the descendant axis, the *classop* will be *dc*, thus `C2` will be: $c_{n-1}.dc$. Therefore, the Descendant-Component and Class-Component combination (i.e. the descendant-or-self branch classes of the previous step) will be the only branch classes evaluated at the last step.

For the first step in the XPath expression, the `Node` or the `NCLTV` relation

must be accessed to identify the first set of context nodes or their proxies, respectively. The index selection process that chooses between these two relations is detailed in §6.4. For each subsequent step (excluding the rightmost), the ClassChain components are used to prune the search space. Then, the `Node` or the `NCLTV` relation is evaluated to locate the target nodes (or their proxies). Finally, the ClassChain components are used one more time (to return the relevant branch classes in `C2`). In other words, component `C4` is repeated *n-1* times, i.e. one less than the number of steps in the expression[1].

Component `C4` contains two variables. The first variable: *indexop* (index options) is populated with `Node` or `NCLTV`. The second variable: x denotes the number of repetitions of `C4`. For example, if there are 5 steps in the expression, `C4` is repeated 4 times. Thus, the x variable will be 1 for the first repetition of `C4`, 2 for the second and so on (up to 4).

6.2.2 Transforming the Initial XPath Step

This *Initial Step* expression (see Template - Figure 6.3) maps directly to the first step in the XPath expression. It filters those tuples associated with aliases i_1 and c_1 (these aliases were discussed for `C4` above). In order to transform the first step in an XPath expression, three additional properties (AxisProperty, NameProperty and TypeProperty) are extracted from the first step in the XPath expression. We will now describe the method that uses these properties to construct the Initial Step expression.

AxisProperty. The XPath axis. The supported axes are: *ancestor, ancestor-or-self, parent, descendant, descendant-or-self and child* (the hierarchical XPath axes).

[1]Square brackets and other BNF symbols (i.e. '+' and '?') are not part of the SQL statement.

NameProperty. The *name* of the target nodes. The name property can also be an XPath wildcard (denoted by '*')

TypeProperty. The *type* of the target node. The type of the target node can be element or attribute.

NextAxisProperty. The XPath axis associated with the next step in left-to-right order. This enables us to identify the branch classes that must be evaluated at the next step. This is crucial in managing the dependency between steps in an XPath expression. Using the `NextAxisProperty`, the class options are as shown in Table 6.5.

Name	Component
C5	WHERE
C6	[i_1.NAME = 'name-prop' AND]?
C7	i_1.TYPE = 'type-prop'
C8	[AND i_1.LEVEL = 0]?
C9	AND $c_1.classop = i_1.CLASS$
C10	[AND EXISTS (<*predicate-statement*>)]*

Table 6.4: The Initial Step Expression Components

The Initial Step expression (*lines 6-9* in the transformation template) contains the components shown in Table 6.4. The first component (`C5`) is the *where* clause, which is required. The where clause filters the *sets* of nodes represented by aliases 'i_1' and 'c_1' (in component `C4`). Component `C6` may be included zero or one times and has one variable (`name-prop`). If the `NameProperty` is a *wildcard* `C6` is omitted (so that nodes of any name are evaluated), otherwise the `name-prop` variable is populated with `NameProperty`. Component `C7` is compulsory and contains a single variable `type-prop`, which is populated with the `TypeProperty`.
An XPath expression that begins with '/' evaluates the document node at the first step. The document node is the only node that resides at *level*

classop	Selected when...
ac	`NextAxisProperty` is the: *descendant*, *descendant-or-self*, or *child* axis.
dc	`NextAxisProperty` is the: *ancestor*, *ancestor-or-self*, or *parent* axis.

Table 6.5: Populating the Class Option Variable

0 in our system. Thus, in this situation component `C8` is included to ensure that the document node is the only node that is evaluated. If the XPath expression begins with '//', nodes at all levels must be evaluated, i.e. *descendant-or-self* nodes of the document node, thus `C8` is omitted.

Component `C9` is used to set the branch classes that are related to the *set* of nodes located by step one in the XPath expression because their hierarchically-related branch classes will be required at the next step. Component `C9` contains a single variable *classop*, which is populated based on the NextAxisProperty, the class options in Table 6.5.

In XPath, a *predicate filter* removes some of the nodes identified at its associated step and leaves others [68]. An XPath step may have one or more predicate filters. Component `C10` is used for each such filter. As component `C10` is generic to all steps (it appears at *lines 9, 15* and *19* respectively in the Transformation Template) we will detail `C10` later in §6.2.5.

In instances where the XPath expression contains only a single step, the process described in this section and the processes described in §6.2.1 and §6.2.3 do not take place. Instead, processing moves directly to the final step transformation described in §6.2.4.

6.2.3 Transforming Interim Steps

The Interim Step expression is used in the transformation of the second step through to the second last step. With the exception of variable x (the step number) the expression at *line 10* is identical to components `C5-C7`. *Line 14* is identical to `C9` and *line 15* is identical to `C10`. Therefore, it is

not necessary to repeat the explanations. The x variable is incremented for each step. For example, in the second step x will be 2. The components within the interim step expression (*lines 11, 12* and *13* in the template) are shown in Table 6.6.

Name	**Component**
C11	AND $i_x.CLASS = c_{x-1}.classop$
C12	[AND i_x.LEVEL *equalop* i_{x-1}.LEVEL [*adlevel*]?]?
C13	[i_x.PRE *equalop* $i_{lnaStep}$.PRE $\wedge$ i_x.POST *equalop* $i_{lnaStep}$.POST]?

Table 6.6: The Interim Step Expression Components

classop	Selected when...
dc	AxisProperty is the: *descendant*, *descendant-or-self*, or *child* axis.
ac	AxisProperty is the: *ancestor*, *ancestor-or-self*, or *parent* axis.

Table 6.7: New Class Options

Component C11 locates the ClassChain components that must be examined. In C11, variable x is the step number and *x-1* denotes the previous step number. In this instance, the *classop* variable is populated based on the AxisProperty as shown in Table 6.7. Component C12 is required if the AxisProperty is the parent or child axis (to differentiate between ancestor/parent and descendant/child axes respectively). Additionally, C12 is required when the *indexop* is the NCLTV relation, in which case C13 is omitted. Based on the current XPath axis, the *equalop* in C12 is populated as shown in Table 6.8.

In addition to the *equalop* variable in C12, the variable *adlevel* (additional level constraint) is added for the parent and child axes. If the axis is parent the *adlevel* variable is populated with *-1*; for the child axis it is populated with *+1*. If the *indexop* is the Node relation, C13 must be included, but C12 omitted for all but the parent and child axes. In C13, *lnaStep* is the number of the last step that evaluated the Node relation (the *lnaStep* is detailed in

Axis	**Option**
ancestor	‘$<$’
ancestor-or-self	‘$\leq$’
descendant	‘$>$’
descendant-or-self	‘$\geq$’
parent	‘$=$’
child	‘$=$’

Table 6.8: Equality Options for Component `C12`

§6.4). The two equality option (`equalop`) variables in `C13` are shown, with ordering, based on the XPath axis in Table 6.9.

Axis	**Option**
ancestor	‘$<$’ ‘$>$’
ancestor-or-self	‘$\leq$’ ‘$\geq$’
descendant	‘$>$’ ‘$<$’
descendant-or-self	‘$\geq$’ ‘$\leq$’
parent	‘$<$’ ‘$>$’
child	‘$>$’ ‘$<$’

Table 6.9: Equality Options for Component `C13`

6.2.4 Transforming the Final XPath Step

The Leaf Opening and Leaf Closing expressions are used in the transformation of the final (or leaf) step in the XPath query expression. This final step ($step_n$) is always the rightmost step in the XPath expression. In the *leaf opening expression*, with the exception that alias i_1 is replaced with $step_n$, *line 2* contains components `C5`, `C6` and `C7` (explained earlier). Similarly, by substituting the same aliases ($step_n$ for i_1), *line 3* is identical to `C8`.

The distinct components in the leaf opening expression are `C14` and `C15` (Table 6.10). Component `C14` is static (i.e. it contains no variables) and selects all tuples from the `Node` relation (the *result set*). If there is more than one step in the expression, the static component `C15` is included to ensure that the branch classes identified at the preceding step are the only

ones that are evaluated in $step_n$. This has the effect of providing a final pruning phase at the point of generating the final result set.

Name	**Component**
`C14`	`SELECT * FROM` Node $step_n$
`C15`	`AND` $step_n$.CLASS IN (
`C16`	)
`C17`	`ORDER BY` PRE

Table 6.10: Leaf Path Expression Components

The second requirement for transforming $step_n$ is to construct the *leaf closing expression*. Within the leaf closing expression, *lines 16* and *17* are identical to `C12` and `C13` respectively, with the exception that each occurrence of alias i_x is replaced with alias $step_n$ and its distinct components are `C16` and `C17` (Table 6.10). Component `C16` terminates the sub-query (which begins at `C15`) and `C17` ensures that the result's nodes are returned in *document order*, as required by the XPath [66] specification.

6.2.5 Transforming XPath Predicate Filters

Within the main transformation template, *lines 9, 15* and *19* contain an `EXISTS` sub-expression that filters nodes based on XPath predicates. In XPath, a predicate filter contains a nested XPath expression, i.e. it consists of a number of steps and each step has an *Axis*, *NodeTest* and zero or more (sub) *Predicates*.

The <*predicate-statement*> (*lines 21-31* in the transformation template) differs from the main template in two ways. Firstly, all of the alias names are given the prefix p (e.g. $pstep_n$) so that alias names in the sub-statements do not conflict with the alias names in the main statement. Secondly, each line in the <*predicate-statement*> that refers back to a line in the main template is identical to that line, with the exception of the prefix on each alias. The new components within the <*predicate-statement*> are shown in Table 6.11.

Name	Component
C18	SELECT *null*
C19	FROM *indexop* $pstep_n$
C20	SELECT $pc_{n-1}.classop$
C21	FROM CLASS pc_1
C22	WHERE pc_1.classop = indexIn.CLASS

Table 6.11: Components within the <*predicate-statement*>

Component C18 is required and does not contain a variable. In component C18, *null* is selected as we do not require the result set from the subexpression (this is the approach used by the authors in [25]). Component C19 is also required and contains the single variable *indexop*. This forms the index selection process, described later in this Chapter, and is a crucial component of the optimisation. Component C20 is the same as C1 and C2, which were explained earlier.

Lines 6-9 do not appear in the predicate statement as these are used to identify the first set of context nodes for the XPath expression. Conversely, in an XPath predicate filter the nodes that must be filtered have already been identified; these nodes are the context nodes for the first step in the predicate filter. The ClassChain components that are associated with these context nodes are identified using component C21. The addition of C21 is the only difference between *line 25* and *line 5*.

The ClassChain components associated with the context nodes at which the predicate is applied are identified using C22. The *classop* variable (in C22) is populated using the `AxisProperty` from the step to which the predicate filter applies and the class options from Table 6.7. The *indexIn* variable is the *alias* for the relation (which will be `Node` or `NCLTV` based on *indexop*) at the step to which the filter applies. For example, if the filter applies to step three in the XPath expression, `indexIn` will be: i_3.

6.3 Sample Transformation

We will now present a sample transformation to illustrate how the transformation template is populated for an XPath query (Example 12).

Example 12

/site/closed_auctions/closed_auction//keyword

6.3.1 Transforming the Generic Expression

The following property is required to populate the generic expression:

- `LastAxisProperty` is the descendant axis, therefore *dc* (descendant or self branch classes) is selected.

Name	Component
C1	SELECT
C2	$c_3.dc$
C3	FROM
C4	NCLT i_1, AD_REL c_1, NCLT i_2, AD_REL c_2, NCLT i_3, AD_REL c_3

Table 6.12: Generic Expression Components

The components in the Generic Expression are populated as shown in Table 6.12. Components `C1` and `C3` are constant (do not contain variables). The *classop* variable in `C2` is populated using the `LastAxisProperty` and the options in Table 6.3. Component `C4` contains three variables: *indexop* is populated based on the index selection process described later in §6.4; CLASS will be the `PC_REL` (if the axis is parent or child and the optional Parent and Child ClassChain components are available) or else it will be the `AD_REL` relation; variable x is the step number. For our current purposes, it is assumed that the Parent and Child ClassChain components are not available, therefore `AD_REL` is used. As there are four steps in the expression, component `C4` is repeated three times.

6.3.2 Transforming the Initial XPath Step

To transform the first step in the XPath expression, the following three properties are required:

- NameProperty = site.
- TypeProperty = 1 (element).
- NextAxisProperty is the child axis, therefore *classop* variable is *ac* (ancestor-or-self branch classes).

Name	Component
C5	WHERE
C6	i_1.NAME = 'site' AND
C7	i_1.TYPE = 1
C8	AND i_1.LEVEL = 0
C9	AND $c_1.ac$ = $i_1.class$
C10	~~AND EXISTS (<*predicate-statement*>)~~

Table 6.13: Initial Step Expression Components

The components within the Initial Step expression are populated as shown in Table 6.13. The NameProperty populates the *name-prop* variable in C6; the TypeProperty populates the type-prop variable in C7 and the NextAxis-Property populates the *classop* variable in C9. There are no predicate filters at the initial step, therefore C10 is omitted.

6.3.3 Transforming Interim Steps

There are two interim steps in Example 12, which are now described.

Transforming the First Interim Step

To transform the first interim step, the following four properties are required:

- AxisProperty = is the child axis, therefore *classop* variable is *dc* (descendant-or-self branch classes).

- NameProperty = closed_auctions.
- TypeProperty = 1 (element).
- NextAxisProperty is the child axis, therefore *classop* variable is *ac* (ancestor-or-self branch classes).

Name	Component
`C5`	`AND`
`C6`	i_2.NAME = ‘closed_auctions’ `AND`
`C7`	i_2.TYPE = 1
`C11`	`AND` $i_2.class = c_1.dc$
`C12`	`AND` i_2.LEVEL = i_1.LEVEL+1
`C13`	~~i_2.PRE *equalop* $i_{lnaStep}$.PRE $\wedge$ i_2.POST *equalop* $i_{lnaStep}$.POST~~
`C9`	`AND` $c_2.ac = i_2.class$
`C10`	~~`AND EXISTS` (<*predicate-statement*>)~~

Table 6.14: First Interim Step Expression Components

As stated when describing the transformation process, components `C5`, `C6` and `C7` are repeated for each interim step. The only difference from their earlier description is that the step number, NameProperty and TypeProperty are updated. Component `C11` is populated by selecting the branch classes associated with the AxisProperty; in `C12` the x variable is populated with the current step number (step 2) and the previous step number (step 1) respectively. Component `C13` is omitted as there were no previous base data evaluations (see index selection §6.4 for more details). Component `C9` is populated with the NextAxisProperty (as it was in the initial step transformation), and the x variable is populated with step 2. There are no predicate filters at this step, thus component `C10` is omitted.

Transforming the Second Interim Step

To populate the second interim step the following properties are used:

- AxisProperty = is the child axis, therefore *classop* variable is *dc* (descendant-or-self branch classes).

- NameProperty = closed_auction.
- TypeProperty = 1 (element).
- NextAxisProperty is the descendant axis, therefore *classop* variable is *ac* (ancestor-or-self branch classes).

Name	Component
`C5`	`AND`
`C6`	i_3.NAME = 'closed_auction' `AND`
`C7`	i_3.TYPE = 1
`C11`	`AND` $i_3.class = c_2.dc$
`C12`	`AND` i_3.LEVEL = i_2.LEVEL+1
`C13`	~~i_3.PRE *equalop* $i_{lnaStep}$.PRE ∧ i_3.POST *equalop* $i_{lnaStep}$.POST~~
`C9`	`AND` $c_3.ac = i_3.class$
`C10`	~~`AND EXISTS` (<*predicate-statement*>)~~

Table 6.15: The Second Interim Step Expression Components

The only difference between the first and the second interim steps is that: (1) the step numbers are updated, where x is now step 3 and *x-1* is step 2 and (2) The NameProperty is now *closed_auction* rather than *closed_auctions*.

6.3.4 Transforming the Final XPath Step

To populate the final step expression, the following properties are used:

- AxisProperty = is the descendant axis, therefore *classop* variable is *dc* (descendant-or-self branch classes).
- NameProperty = keyword.
- TypeProperty = 1 (element).

In the Leaf Opening expression, components `C5`, `C6` and `C7` are updated with the relevant step numbers and properties as shown in Table 6.16. Component `C8` is not required because it is not a single step XPath expression. Component `C14` completes the Leaf Opening expression.

Name	Component
C14	SELECT * FROM Node $step_n$
C5	WHERE
C6	$step_n$.NAME = 'keyword' AND
C7	$step_n$.TYPE = 1
C8	~~[AND $step_n$.LEVEL = 0]?~~
C15	AND $step_n$.class IN (
C12	AND $step_n$.LEVEL > i_3.LEVEL
C13	~~i_3.PRE *equalop* $i_{lnaStep}$.PRE $\wedge$ i_3.POST *equalop* $i_{lnaStep}$.POST~~
C16	)
C17	ORDER BY PRE

Table 6.16: Leaf Opening and Closing Components

In the Leaf Closing expression component C15 is constant. In component C12, variable x is populated with the previous step number (step 3). Component C13 is not reqiured because there are no provious base data evaluations (see index selection §6.4) for more details. Finally, components C16 and C17 are static and C17 ensures that the result set is returned in XML *document order*. The completed SQL expression is shown in Figure 6.4.

```
-- Begin Leaf Opening Expression
SELECT * FROM Node stepN

 WHERE stepN.NAME = 'keyword'
 AND   stepN.TYPE = 1
 AND stepN.class IN (
 -- End Leaf Opening Expression

    -- Begin Generic Expression
    SELECT c3.dc FROM
        NCLT i1, AD_REL c1, NCLT i2, AD_REL c2, NCLT i3, AD_REL c3
    -- End Generic Expression

        -- Begin Initial Step Expression
        WHERE i1.NAME = 'site' AND i1.TYPE = 1
        AND i1.LEVEl = 0
        AND c1.ac = i1.class
        -- End Initial Step Expression

        -- Begin First Interim Step Expression
        AND i2.NAME = 'closed_auctions'
        AND i2.TYPE = 1
        AND i2.class = c1.dc
        AND i2.LEVEL = i1.LEVEL+1
        AND c2.ac = i2.class
        -- End First Interim Step Expression

        -- Begin Second Interim Step Expression
        AND i3.NAME = 'closed_auction'
        AND i3.TYPE = 1
        AND i3.class = c2.dc
        AND i3.LEVEL = i2.LEVEL+1
        AND c3.ac = i3.class
        -- End Second Interim Step Expression

        -- Begin Leaf Closing Expression
        AND stepN.LEVEL > i3.LEVEL

)

ORDER BY PRE
-- End Leaf Closing Expression
```

Figure 6.4: The Completed SQL Expression

6.4 Index Selection

Our optimization strategy includes a number of rules that determine when the index can be used, as the alternative is to access the base data to compare individual nodes. Therefore, the goal of the index selection phase is to maximize usage of the index. This is achieved though a series of rules. We now describe how the query process decides when the index can be used and when the base data must be evaluated (that is, we describe the decision making process for selecting the `indexop` variable in the transformation).

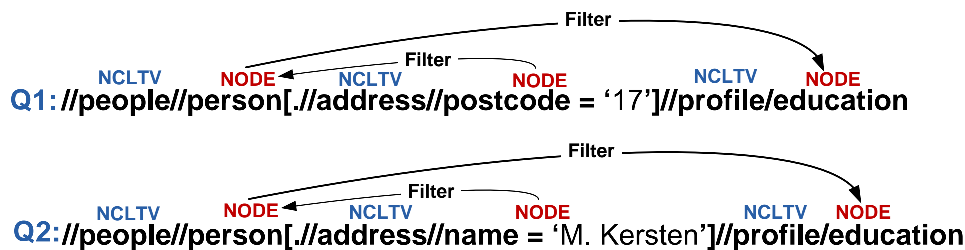


Figure 6.5: Index Selection

6.4.1 Base Index Selection Rules

When an XPath step evaluates a text value that has not been classified, the base data must be accessed. For all Twig queries that do not evaluate a text value (that has not been classified), the following index selection rules are used:

Rule 1. If the step being processed is the *rightmost* step in each XPath expression, then the base data (the `Node` relation) must be used to evaluate the expression. This rule supersedes all other rules and remaining rules are applied only if Rule 1 does not hold.

Rule 2. If the step does not evaluate a text value or if the step evaluates a classified text value, then the index is used.

Rule 3. For any step that evaluates a non-classified text value, e.g. `//postcode = '17'` (`Q1`), the base data is used.

Rule 4. For a step that contains a predicate filter that evaluates a non-classified text value or contains the following or preceding axis, the base data is used. For example, *step two* (`Q2`) contains a predicate filter that evaluates a text value.
That completes the base rules, next we present the advanced index selection rules.

6.4.2 Advanced Index Selection Rules

Before presenting the index selection process, we must introduce the concept of a *context shift*.

Definition 9 *A Context Shift occurs when it cannot be guaranteed that the context nodes associated with a step are within the (pre/post) range of the previous predicate filter.*

For the index selection process it is crucial that the nodes that must be evaluated at the current step are within the range of the previous predicate filter (Definition 9). This is because this filter (which requires a base data evaluation) does not have to be applied to steps if it can be guaranteed that they are within the range of the previous predicate filter. A context shift can only occur after the first step to which *rule 4* applies. For example, in `Q2` (Figure 6.5), the predicate filter that is applied to step two is not applied to step three, but it is applied to step four. Using XPath logic, a base data evaluation is avoided at step three because it can be determine that all nodes that could possibly be evaluated at step three are within the range of the predicate filter (there is no context shift).

The following three cases result in a context shift:

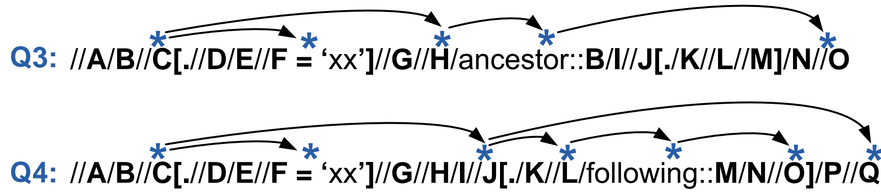


Figure 6.6: Index Selection (Context Shift)

Case 1. A step contains an axis in an opposite hierarchical direction to the previous step, for example, a step that contains the descendant or child axis followed by a step that contains ancestor or parent. This is exemplified in the sixth step (`/ancestor::B`) of `Q3` (Figure 6.6), that is, the fifth step contains the descendant-or-self axis and the sixth step contains the ancestor axis.

Case 2. A step that contains the *preceding* or *following* axis is encountered. For example, if we replaced the ancestor axis in the sixth step of `Q3` with following or preceding, the result would still require a context shift.

Case 3. *Rule 4* (of the base index selection rules) applies to a step. This is exemplified in the seventh step of `Q4` as the predicate filter associated with this step contains the following axis.

In Figure 6.6, an asterisk above a step denotes a base data evaluation. When a base data evaluation is required, the unique node labels (e.g. pre/post) of the nodes must be compared. For all other steps it is sufficient to evaluate *name*, *class*, *level*, *type* and *value* (i.e. the `NCLTV` index may be used instead). Each time a context shift occurs based on *case 1* or *case 2*, a *join* operation (that compares node labels) is performed between the step immediately preceding the step to which the context shift applies and the step that required the previous base data evaluation. For example, the join between the third and fifth steps in `Q3`. Next, the step to which the context shift applies is joined with the step to its immediate left, such as, the join between

the fifth and sixth steps in `Q3`.

If a context shift occurs based on *case 3*, the step to which this context shift applies is joined with the step that required the previous base data evaluation - the Last Node Access Step (lnaStep) variable described in the transformation process. In `Q4`, a context shift (based on *case 3*) occurs at step seven, therefore it is joined with step three. Finally, the rightmost step is always joined with the Last Node Access Step (if such a step exists).

6.5 Summary

In this section, we discuss the integrity of query results and provide a summary of the chapter.

6.5.1 Integrity Checking for Transformation Process

In §6.3, we described our Transformation Template, a construct used to construct SQL queries from XPath expressions. This process is designed to benefit from concepts such as partitions, classification of partitions and indexing low selectivity text nodes. In §6.3, we provided a detailed workflow for a sample XPath expression. Our final transformation process is a result of many iterations of converting XPath expressions to SQL expressions suitable for our indexing system. Initial transformations were fairly simple [46], different optimisations were then introduced in [47] and in [48]. At each point in the development of the transformation algorithms, we developed an integrity check to ensure that our transformations were correct.

Our early algorithms (before partitions) were based on the work of [29]. As such, we used the transformation algorithms and the processes of established researchers in the area. After we developed our own approach, we adopted a two-phase quality check. Firstly, we compared the results from each new iteration with the results from the previous iteration. Secondly, we compared

with an existing implementation (MonetDB/XQuery), by running identical queries on both systems. In this phase, we checked that: a) the node counts were identical, and (b) the name, type (element/attribute) were also identical, as the SQL query is based on '*name*' and 'type'. In both phases, the check was performed on all 51 queries presented in the experiments chapter.

6.5.2 Final Summary

In this chapter, we described a deployment of the BranchClassIndex within a relational database. To evaluate XPath expressions using an index that is deployed in a relational database, an XPath-to-SQL transformation process is required to transform XPath expressions to their SQL equivalents. Thus, we described the large subset of the XPath language that our system covers and described the XPath-to-SQL transformation process in detail and provided a worked example. Finally, as some text values were not included in the classification process that was described in Chapter 5, there are instances where the `NCLTV` covering index cannot be exploited to optimise a step. Therefore, this Chapter is completed with a description of the index selection process that determines whether the index can be used for an XPath step and when the base data must be evaluated because text values appear in the XPath expression.

Chapter 7

Experiments

This chapter provides details of the experiments used to evaluate the BranchClassIndex. In doing so, we evaluate the performance of a traditional node-based approach [31]. However, as the evidence in this evaluation shows, node-based approaches (such as [29, 31]) cannot scale to XML datasets of the size used in our study. Thus, in a separate evaluation using a relatively small dataset and queries taken from XPathMark benchmark [67], we evaluate node-based approaches at the end of the chapter in §7.5.

This chapter is structured as follows: §7.1 describes the evaluation method; in §7.2 we specify low cardinality text values empirically; in §7.3, we evaluate the real world XML case study. For reasons of experimental repeatability [44,45], we evaluate the same XML datasets and queries as other researchers in §7.4; finally, in §7.5, we evaluate node-based approaches.

7.1 Evaluation Method

In the main evaluation, the BranchClassIndex is compared to one of the more recent path-based approaches [24], which we refer to as the PathBasedIndex. MonetDB/XQuery [12] is also evaluated because it uses the native Staircase Join algorithm [32] and because it was evaluated in [24] (the path-based

approach used in our evaluation). Finally, the commercial XML database, SQL Server 2008, is evaluated because it uses the relational XML indexing techniques described in [57], which were subsequently discussed by the authors of [31] (the node-based approach used in our evaluation).

The BranchClassIndex, the PathBasedIndex, SQL Server 2008, and MonetDB/XQuery were deployed on identical servers with a 2.66GHz Intel(R) Core 2 Duo CPU and 4GB of RAM. The BranchClassIndex and the PathBasedIndex were deployed in an *Oracle 11g* relational database. Oracle 11g and MonetDB/XQuery *version 4.34.4* were both deployed on *Fedora 12 Linux* (64bit) platforms; SQL Server was deployed on a *Windows 7* (64bit) platform.

For queries executed across the vendor systems, we call the `count()` function to ensure that any overhead associated with *document reconstruction* [15] is not included in the query response times. This approach was also used in [25] for evaluating the comparative query response times of vendor systems. In order to conduct a balanced evaluation, we also called the `count()` function on the SQL queries executed across the BranchClassIndex and the PathBasedIndex.

Each XPath query used in the experiments was run twenty one times. Using the same approach as [26, 60], the first run was ignored to ensure *hot cache* response times and the remaining twenty runs were averaged to provide the final result in milliseconds. A *timeout* of ten minutes was placed on each query to allow us to perform the evaluation in a reasonable amount of time. Each query that took longer than ten minutes is tagged as *>10mins*. Additionally, to calculate average query response times, each query that is tagged as *>10mins* is counted as 600,000 milliseconds.

7.1.1 Implementation and Deployment Decisions

We made the following practical decisions when implementing the BranchClassIndex and the PathBasedIndex:

- For performing node comparisons, *pre/post* labels are used in both the BranchClassIndex and the PathBasedIndex (both approaches are node label independent).

- Schema-oblivious [24] node storage was used in both approaches as the BranchClassIndex is a schema-oblivious approach. The authors of the PathBasedIndex described a schema-aware and a scheme-oblivious version of their PathBasedIndex. We chose schema-oblivious node storage because schema-aware approaches may incur inefficiencies when *wildcards* exist in XPath queries, i.e. *SQL Splitting* [24] is required.

- In both approaches, attribute nodes are stored as separate nodes that are differentiated from element nodes using the *type* column. The authors of the PathBasedIndex map attribute nodes to relational columns.

- For the BranchClassIndex and the PathBasedIndex, a B-tree index was built on each column of each relation.

- Oracles's *cost based optimiser* is allowed to chose the best *query execution plan.* In other words, we did not supply *optimiser hints* (to manually control query executions plans).

The following options where selected when deploying SQL Server and MonetDB/XQuery:

- In SQL server, we built the *primary index*, and the *secondary indexes* `PATH` and `VALUE`, which are the optimisation techniques described in [57].

- In MonetDB/XQuery, the *read only* storage option was selected in order to exclude any performance overhead associated with updateable storage options.

7.1.2 Query Categories

The following five query categories enable us to be precise about the type of query each approach evaluates most efficiently.

$\mathbf{QC_1}$. Linear Path Queries. This type of query does not contain predicate filters.

$\mathbf{QC_2}$. Twig Queries without Text Values. This type of query does not evaluate text values, but contains predicate filters.

$\mathbf{QC_3}$. Twig Queries with Low Selective Text Values. This type of query contains predicated filters and evaluates text values that have *low selectivity*.

$\mathbf{QC_4}$. Twig Queries with High Selective Text Values. This type of query contains predicate filters and evaluates text values that have *high selectivity*.

$\mathbf{QC_5}$. Single Step Path Fragment Queries. This type of query does not contain a primary path fragment (PPF), as defined for the PathBasedIndex [24], that spans more than one step in the XPath expression. This query category is used to highlight a class of XPath queries in which path-based approaches do not improve performance.

7.2 Specifying Low Selectivity Text Values

In Chapter 5, we presented two algorithms that identify low selectivity text values that are suitable for classification (i.e. suitable for indexing). The `IdentifyRangeBasedCandidates` algorithm identifies nodes that can have

all of their values indexed, therefore range based evaluations (e.g. year > '1985' and year < '2000') can be performed in the index. If a node, such as *year*, contains a large number of distinct text values, all of its text values may not be suitable for indexing (because of increased index size). In which case, the `IdentifyEqualityBasedCandidates` algorithm may identify individual text values as being suitable for indexing, but only equality evaluations (e.g. year = '1985') are for such values.

Both text value identification algorithms depend on *time* `T`; the number of *context nodes* (`CN`); and the number of *initial target nodes* (`TN`): `T = CN * TN`, where `T` was determined after a number of experiments.

As a starting point, we performed a number of experiments to determine when `T` is 1000ms (one second). For this purpose, we created temporary node repositories of various sizes and counted the maximum number of individual node comparisons that could be performed based solely on *pre/post* node labels in less than one second. Using the same system on which the BranchClassIndex was deployed (described earlier), we found that `T` is 1000ms when `CN` is 3100 and `TN` is 3100. In other words, (3100 * 3100) is the maximum number of *pre/post* node comparisons that can be performed in less than one second.

However, the threshold of `N` equal to 3100 acts only as a guideline due to the fact that, if there are a very large number of text values that occur more than `N` times, then it can lead to a degradation in query performance. Later on, we will demonstrate the performance of the BranchClassIndex for a number of text value subsets.

7.3 The Bicycle Rental Dataset

A real world dataset with genuine user requirements was chosen to determine the wider impact of our work. One of our research group's other projects includes a smart city project, which provided us with this dataset and a

No.	Cat	XPath Query	Nodes
Q01	QC_1	Return all information recorded for stations in Nantes. /bikes/city/Nantes/stations/station	1,411,451
Q02	QC_1	Return the number of bikes that are free across all stations, and all days, in Dublin. /bikes/city/Dublin/stations/station/free	634,320
Q03	QC_2	Return all stations, from all cities, that have bicycle availability information. //city//stations[./station/available]	191,680
Q04	QC_2	Return all stations, from all cities, that have weather information including: wind direction and speed; and the date and time it was received. //city//stations[.//weather/time][.//weather//wind/direction][.//weather//wind/speed]	191,680
Q05	QC_3	Return all information regarding stations in Luxembourg in which there were no free bikes. //Luxembourg/stations[./station/available = '0']	13,046
Q06	QC_3	Return the *id* (identifier) of each station that had no bikes available. //stations/station[./available = '0']/id	429,585
Q07	QC_3	Return the cities that had a wind speed greater than 6 miles per hour. //stations[.//wind/speed > '6']/parent::*	145
Q08	QC_5	Return the stations that had a wind direction of 40. //direction[.= '40']/ancestor::stations /station	192,755
Q09	QC_4	Return all entries for Lyon on the date *01/06/2010*. //Lyon[./@day = '01'][./@month = '06'][./@year = '2010']	1
Q10	QC_4	Return the wind chill in Lyon on the date *01/06/2010*. //Lyon[./@day = '01'][./@month = '06'][./@year = '2010']//chill	642

Table 7.1: Bicycle Rental Queries

number of specified queries [49]. A copy of the bicycle rental dataset (2.06 GB) is available on our website[1] and the ten queries used in the evaluation are shown in Table 7.1.

7.3.1 Query Analysis

The BranchClassIndex does not contain text values (i.e. the branch classification process was not extended to include text values). For each of the ten queries, Table 7.4 shows the query response time for the BranchClassIndex (BCI), the PathBasedIndex (PBI), MonetDB/XQuery (MDB) and SQL Server 2008 (SQLS). For each query category, the average query response time is shown and the fastest time is highlighted with the colour blue. Additionally, the total average query response time across all query categories is shown and again the fastest time is highlighted in blue.

For the bicycle rental dataset, Table 7.5 shows: the number of nodes in the `Node` relation; the number of actual branch classes; the number of *rows* in the `NCLTV` relation; and the number of branch class relationships in the `PC_REL` and `AD_REL` relations.

[1] *http://computing.dcu.ie/~isg/BicycleRental*

Query	Cat	BCI	PBI	MDB	SQLS
Q01	QC$_1$	1,130ms	3,110ms	5,662ms	258ms
Q02	QC$_1$	1,084ms	1,070ms	3,152ms	166ms
AVG		**1,107ms**	**2,090ms**	**4,407ms**	**212ms**
Q03	QC$_2$	812ms	>10mins	19,498ms	>10mins
Q04	QC$_2$	325ms	>10mins	16,766ms	>10mins
AVG		**569ms**	**600,000ms**	**18,132ms**	**600,000ms**
Q05	QC$_3$	87,385ms	157,090ms	11,890ms	>10mins
Q06	QC$_3$	>10mins	>10mins	108,192ms	>10mins
Q07	QC$_3$	96,143ms	>10mins	107,629ms	>10mins
AVG		**261,176ms**	**452,363ms**	**75,904ms**	**600,000ms**
Q09	QC$_4$	31ms	140ms	162ms	1ms
Q10	QC$_4$	484ms	68,690ms	5,215ms	166ms
AVG		**258ms**	**34,415ms**	**2,688ms**	**84ms**
Q08	QC$_5$	**>10mins**	**>10mins**	**105,129ms**	**error**
Total AVG		**138,739ms**	**323,010ms**	**38,329ms**	**333,399ms**

Table 7.2: Results for the Bicycle Rental Dataset

NAME	Nodes	Branch Classes	NCLTV	PC_REL	AD_REL
BranchClassIndex	85,965,102	1,067	1,224	2,133	5,032

Table 7.3: Branch Index Statistics

Linear Path Expressions (Q01, Q02)

Queries Q01 and Q02 are linear path queries. Both SQL Server and the PathBasedIndex process all steps in linear path queries simultaneously. Conversely, the BranchClassIndex and MonetDB/XQuery perform step-at-a-times evaluations.

SQL Server (SQLS) performed best overall in this category followed by the BranchClassIndex, PathBasedIndex and MonetDB/XQuery respectively. We believe that SQL Server performed well in this category because its `PATH` index [20,57] is optimised for linear path queries. MonetDB/XQuery performs a Staircase Join [32] between each step in linear path expressions. The fact that SQL Server and the PathBasedIndex outperformed MonetDB/XQuery indicates that, for linear path queries, path-based approaches are more ef-

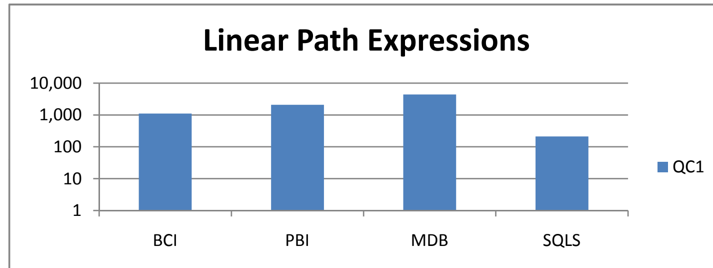


Figure 7.1: Average Linear Path Expression Performance

fective for this dataset.

Similar to MonetDB/XQuery, the BranchClassIndex also performs a join at each step in the XPath expression. However, the size of the BranchClassIndex is small (see Table 7.5) for this dataset (in comparison to later datasets), which may be why it outperformed the PathBasedIndex. Later in this chapter, we show that (when evaluating other datasets) the PathBasedIndex can in some instances outperform the BranchClassIndex for linear path queries.

Twig Queries without Text Values (Q03 and Q04)

The BranchClassIndex performed best overall for Twig queries that do not evaluate text values (QC_2) - see Figure 7.2. This is because the `NCLTV` covering index can be exploited to optimise all steps in the XPath expression (the rightmost step excepted). In contrast, the PathBasedIndex must perform a costly join between each *primary path fragment* (PPF). For example, query Q03 has two primary path fragments: *//city//stations* and */station/available*, whereas query Q04 has four PPFs. One join operation is required for Q03 and four joins are required for Q04, which leads to a significant performance overhead for the PathBasedIndex. SQL Server's `PATH` index suffers the same drawback. Thus, the PathBasedIndex and SQL Server show very similar query response times and they are the slowest approaches

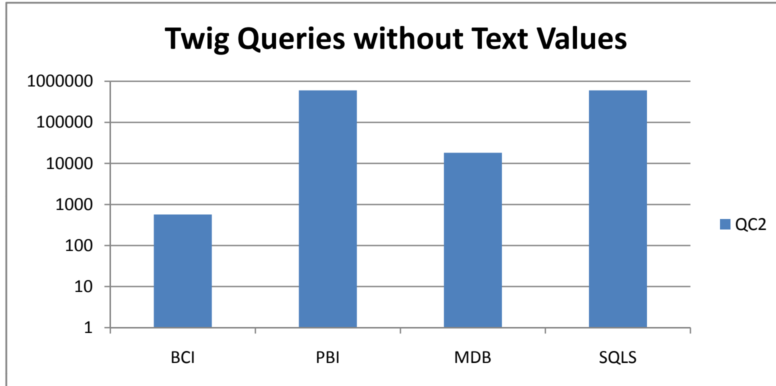


Figure 7.2: Average for Twig Queries without Text Values

for queries in QC_2[2].

MonetDB/XQuery performed better than the PathBasedIndex and SQL Server. This indicates that when node comparisons are required between path fragments, even though MonetDB/XQuery performs more joins (one between each XPath step rather than each path fragment), the Staircase Joins may perform more efficiently overall.

Twig Queries with Low Selectivity Text Values (Q05, Q06, Q07)

For queries in QC_3, the PathBasedIndex and SQL server suffer the same drawback as they did in QC_2 (joins are required between path fragments). The BranchClassIndex cannot exploit the `NCLTV` index at every step because queries in QC_3 contain text values. Before branch classification extensions, the BranchClassIndex must access the base data to evaluate text values (because they do not exist in the index). MonetDB/XQuery performs best overall in this category indicating that, for the bicycle rental dataset, the Staircase Join is more consistent across all Twig queries. However, MonetDB/XQuery took more than 100 seconds to evaluate both Q06 and Q07,

[2]In our earlier work [46], we optimised this inefficiency associated with path-based approaches by extending path fragments to *some* leaf nodes within predicate filters if they have high selectivity and working backwards, thus reducing the number of nodes that must be processed in the join between path fragments. In this work, statistics including the selectivity of nodes are generated and indexed in advance of the query process.

which makes this its least efficient category.

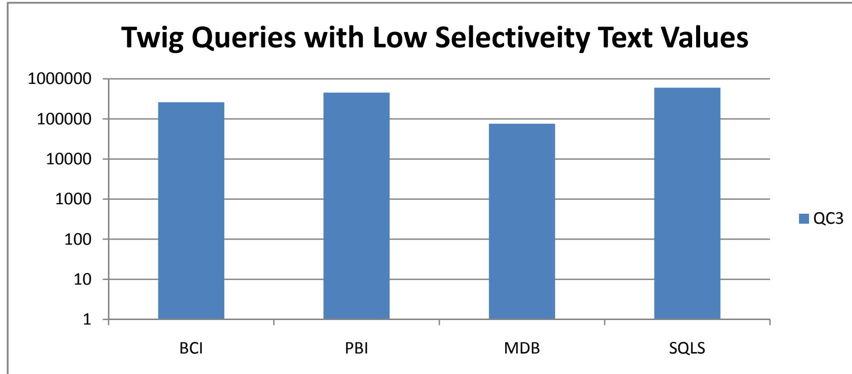


Figure 7.3: Average for Twig Queries with Low Selectivity Text Values

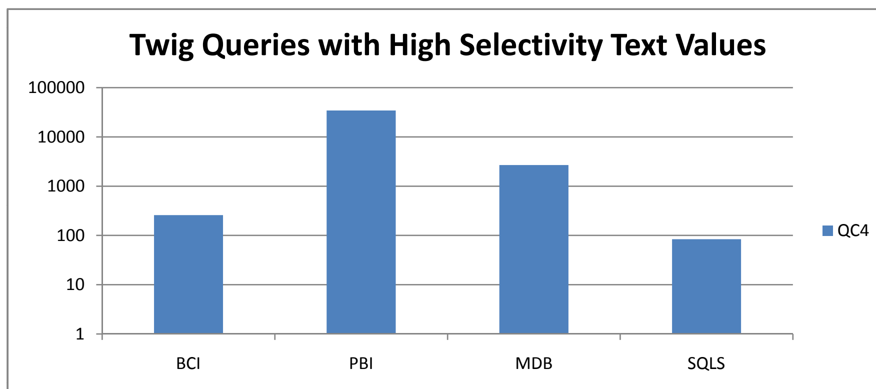


Figure 7.4: Average for Twig Queries with High Selectivity Text Values

Twig Queries with High Selectivity Text Values (Q09, Q10)

Even though the BranchClassIndex does not contain text values, the high selectivity of the text values in QC_4 ensures that few node comparisons. The BranchClassIndex performed better than the PathBasedIndex because step two (`//chill`) in query Q10 requires a base data evaluation in both approaches (to locate the result nodes), but for the BranchClassIndex the query process exploits branch classes to prune the search space. The Branch-ClassIndex outperforms MonetDB/XQuery, indicating that it bypasses a

larger number of nodes during the query process. SQL Server performed best overall, which may be attributed to its `VALUE` index, which optimises text value evaluations (particularly for values that have high selectivity [20]).

Single Step Path Fragment Queries (Q08)

QC_5 is used to illustrate a category of queries in which PathBasedIndex's primary path fragments and SQL Server's `PATH` index cannot be exploited. In other words, *root path* indexes are redundant if each path fragment (such as simple path expression in XRel [69] or primary path fragments in [24]) spans just one step each and thus no optimisation is achieved. The low selectivity of the text value '*40*' in Q08 is responsible for the inefficiency of the BranchClassIndex. However, the PathBasedIndex performs poorly in this category even when the queries do not contain text values. This is because each primary path fragment spans just one XPath step and therefore performs at the same level as node-based approaches (node-based approaches are evaluated later in §7.5). For example, we ran the same query without the text value (`//direction/ancestor::stations/station`) and the PathBasedIndex took more than ten minutes to return a result, whereas the BranchClassIndex took just three seconds.

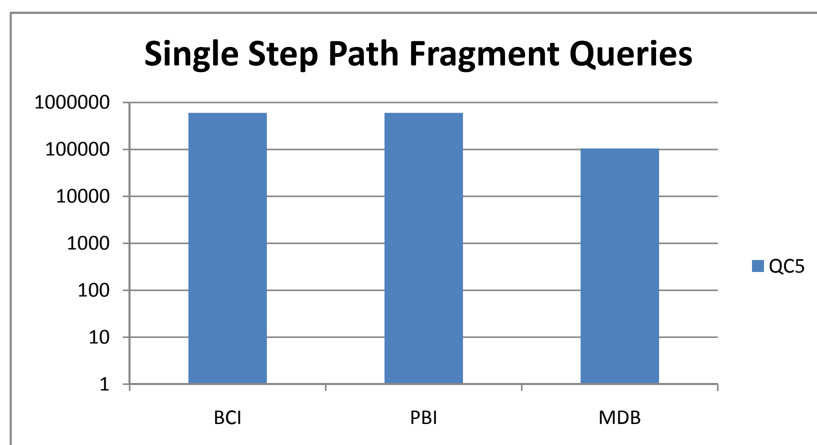


Figure 7.5: Average Single Step Path Fragment Queries

SQL Server does not support the ancestor axis, which is why no result is shown for Q08. MonetDB/XQuery is the most efficient approach for Q08, and combining this with the fact that it is also the most efficient for queries in QC_3 (Twig queries with low selectivity text values), indicates that it is the most efficient approach for Twig queries that contain low selectivity text nodes. There is overlap between QC_3 and QC_5 because Q08 can belong to either category; category QC_5 is only used to illustrate a category of queries that path-based approaches cannot optimise.

Overall Performance

Across all queries, the average performance of each approach is illustrated in Figure 7.6. These averages show that MonetDB/XQuery is the most highly optimised approach overall, followed by the BranchClassIndex, the PathBasedIndex and SQL Server respectively. However, we now show how the BranchClassIndex can be optimised further by classifying text values.

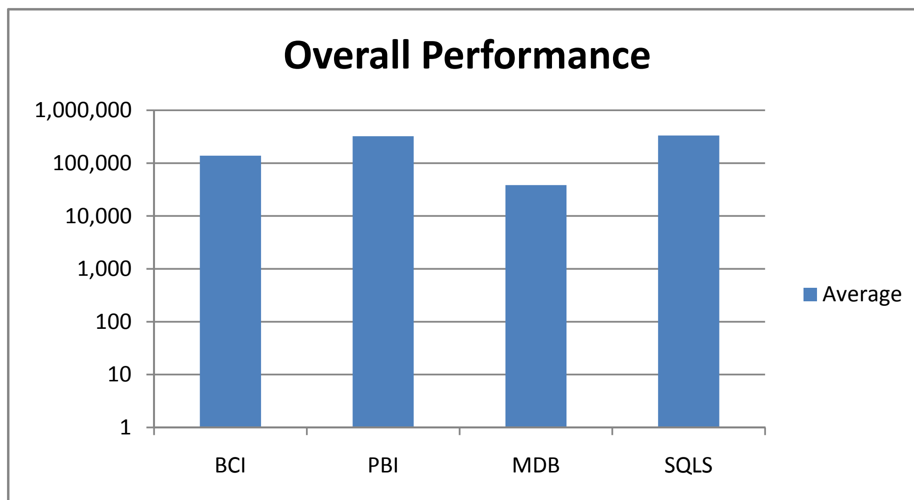


Figure 7.6: Average Performance across all Query Categories

Query	Cat	BCI_1	BCI_2	BCI_3	BCI_4	BCI_5	BCI_6	PBI	MDB	SQLS
Q01	QC_1	1,130ms	1,120ms	5,718ms	12,288ms	1,041mw	4,104	3,110ms	5,662ms	258ms
Q02	QC_1	1,084ms	1,083ms	3,913ms	7,590ms	1,666ms	2,159	1,070ms	3,152ms	166ms
AVG		**1,107ms**	**1,102ms**	**4,816ms**	**9,939ms**	**1,354ms**	**3,132ms**	**2,090ms**	**4,407ms**	**212ms**
Q03	QC_2	812ms	103ms	12,249ms	13,582ms	269ms	317	>10mins	19,498ms	>10mins
Q04	QC_2	325ms	405ms	72,372ms	70,932ms	2,078	3,046	>10mins	16,766ms	>10mins
AVG		**569ms**	**254ms**	**42,311ms**	**42,257ms**	**1,174ms**	**1,682ms**	**600,000ms**	**18,132ms**	**600,000ms**
Q05	QC_3	87,385ms	87,972	2,002ms	2,404ms	157ms	245	157,090ms	11,890ms	>10mins
Q06	QC_3	>10mins	>10mins	14,479ms	8,232ms	1,180ms	1,241	>10mins	108,192ms	>10mins
Q07	QC_3	96,143ms	96,143	1,433ms	1,779ms	63ms	102	>10mins	107,629ms	>10mins
AVG		**261,176ms**	**261,371ms**	**5,971ms**	**4,138ms**	**466ms**	**529ms**	**452,363ms**	**75,904ms**	**600,000ms**
Q09	QC_4	31ms	147ms	197ms	156ms	8ms	369	140ms	162ms	1ms
Q10	QC_4	484ms	614ms	11,658	12,056	7,957ms	7,026	68,690ms	5,215ms	166ms
AVG		**258ms**	**381ms**	**5,928ms**	**6,106ms**	**3,983ms**	**3,698ms**	**34,415ms**	**2,688ms**	**84ms**
Q08	QC_5	**>10mins**	**>10mins**	**1,388ms**	**1,391ms**	**2,043ms**	**2,175ms**	**>10mins**	**105,129ms**	**error**
Total AVG		**138,739ms**	**138,758ms**	**12,540ms**	**13,041ms**	**1,646ms**	**2,078ms**	**323,010ms**	**38,329ms**	**333,399ms**

Table 7.4: Bicycle Rental Dataset Results (after Text Value Classification)

7.3.2 Query Analysis after Text Value Classifications

The BranchClassIndex is now evaluated under a number of separate conditions to highlight the benefit of its individual components. The query response times are shown before and after the classification process has been extended to include text values. In addition, the query response times are shown with and without the inclusion of the Parent and Child ClassChain components, which (as discussed in Chapter 5) are optional.

For the bicycle rental dataset, the text value identification algorithms identified a total of 519 low selectivity text values as being suitable for classification. Of these, we classified a large subset (227) for the experiments described in this section (this subset includes the low selectivity text values in query categories QC_3 and QC_5). The following six versions of the BranchClassIndexes are now evaluated:

BCI_1. The BranchClassIndex before classification of text values and including Parent and Child ClassChain Components.

BCI_2. The BranchClassIndex before classification of text values and *not* Parent and Child ClassChain Components.

BCI_3. The BranchClassIndex with 227 text values and including the Parent and Child ClassChain Components.

BCI_4. The BranchClassIndex with 227 text values and not including the Parent and Child ClassChain Components.

BCI_5. The BranchClassIndex with 100 text values and including the Parent and Child ClassChain Components.

BCI_6. The BranchClassIndex with 100 text values and *not* including Parent and Child ClassChain Components.

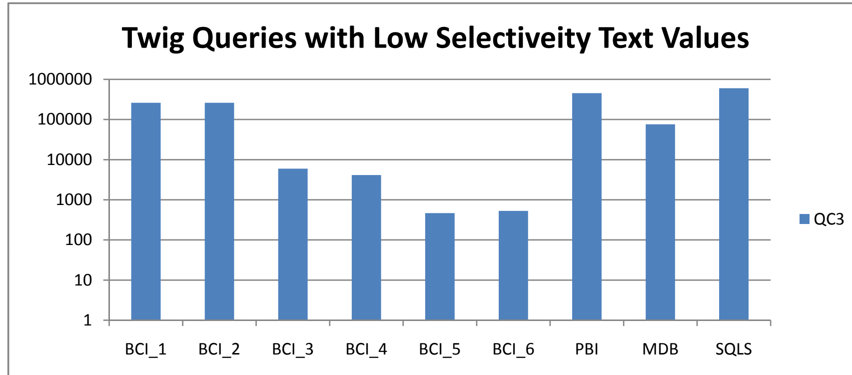


Figure 7.7: Average for Twig Queries with Low Selectivity Text Values

NAME	Nodes	Branch Classes	NCLTV	PC_REL	AD_REL
BCI_1/2	85,965,102	1,067	1,224	2,133	5,032
BCI_3/4	85,965,102	2,303,622	2,351,127	4,607,241	11,583,109
BCI_5/6	85,965,102	143,888	147,503	287,775	699,188

Table 7.5: Branch Index Statistics

For each BranchClassIndex, the variations in the number of branch classes and the size of the `NCLTV`, `PC_REL` and `AD_REL` relations are shown in Table 7.5. These variations are important as they are responsible for the performance deviations described in this section.

Twig Queries with Low Selectivity Text Values (Q05, Q06, Q07)

The performance value of branch classification extensions is clear for queries that evaluate text values that are of low selectivity (QC_3). Figure 7.7 shows that the BranchClassIndex performs significantly better when text values are indexed (BCI_3, BCI_4, BCI_5, BCI_6) than when they are not (BCI_1, BCI_2). Even when a large subset (227) of text values are classified (BCI_3, BCI_4) significant performance gains are achieved. However, when fewer text values are classified (100), the BranchClassIndex performs better.

In our earlier assessment of queries in category QC_3, MonetDB/XQuery performed best overall and queries that evaluate text values that are of low

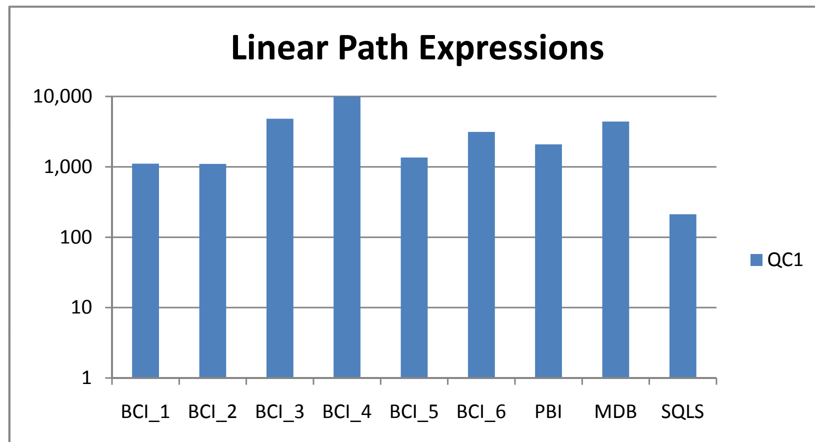


Figure 7.8: Average Linear Path Expression Performance

selectivity were identified as worst case queries for the BranchClassIndex. In contrast, after classification extensions, the BranchClassIndex is the most highly optimised approach even when a large subset of text values are classified (see Figure 7.7).

Linear Path Expressions (Q01, Q02)

Linear path queries only evaluate the base data at the rightmost step (to locate the result set). For this reason, extending the classification process to include text values cannot optimise linear path queries. In fact, they can incur a performance overhead. This is illustrated in Figure 7.8, which shows that BranchClassIndex before the inclusion of text values (BCI_1 and BCI_2) performs better than after their inclusion (BCI_3, BCI_4, BCI_5, BCI_6). Additionally, Figure 7.8 shows that when the classification process includes 100 text values (BCI_3, BCI_4) it performs better than when it contains 227 text values (BCI_5, BCI_6). This is because the number of branch classes increases, and in particular, the `PC_REL` and `AD_REL` relations grow significantly in size. There is therefore a trade off between the number of text values that are included in the classification process and the performance overhead they incur for linear path queries. However, a subset of 100 text

values (BCI_5) performs almost as well as the BranchClassIndex before the extensions.

Note also that the optional Parent and Child ClassChain components minimise the effect of the increased index size (e.g. BCI_3 performs better than BCI_4 and BCI_5 performs better than BCI_6), whereas when text values are not classified these components do not have a query performance benefit (BCI_1 and BCI_2 perform equally well). The reason for this is that the Parent and Child ClassChain components are stored in the `PC_REL` relation. An examination of the size of the `PC_REL` relation in contrast to the `AD_REL` relation (Figure 7.5) shows a much larger differentiation when text values are classified. For example, for BCI_1 the `AD_REL` relation contains around 3000 more relationships than the `AD_REL` relation. In contrast, for BCI_3 the differential is around 7 million relationships, thus the Parent and Child ClassChain components have a significant search space pruning effect.

Twig Queries without Text Values (Q03, Q04)

Figure 7.9 shows that the results for Twig queries that do not contain text values are similar to those for linear path queries. Again, the reason for this is that by specification, queries in QC_2 do not evaluate text values. Therefore, the inclusion of text values in the index has no value and may incur a performance overhead due to the increased index size.

The only difference shown between queries in QC_1 and those in QC_2 is that the benefit of the Parent and Child ClassChain components in QC_2 is not significant. The reason for this is that in total, the queries in QC_1 contain eleven parent or child axes, whereas those in QC_2 contain only five. In other words, the particular queries in QC_2 cannot exploit the Parent and Child ClassChain components as often as those in QC_1.

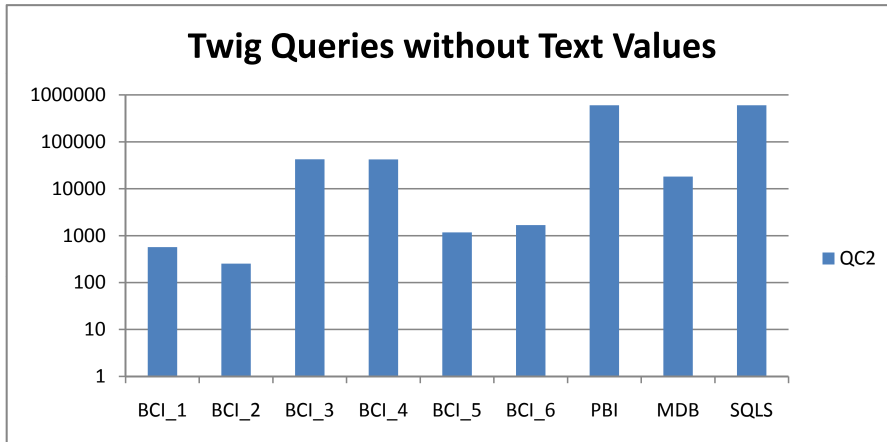


Figure 7.9: Average for Twig Queries without Text Values

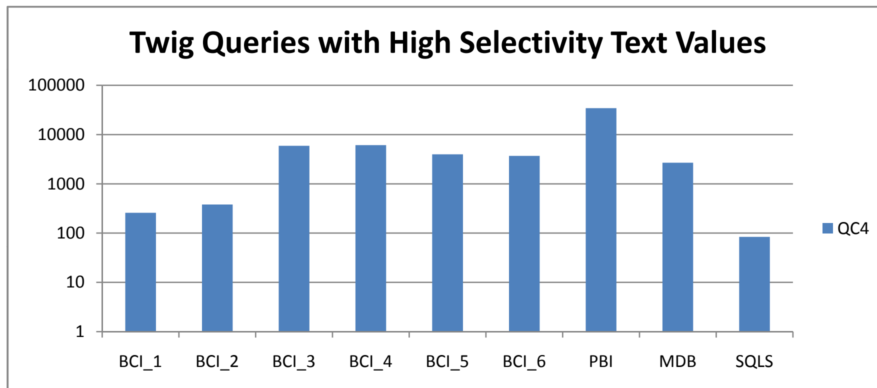


Figure 7.10: Average for Twig Queries with High Selectivity Text Values

Twig Queries with High Selectivity Text Values (Q09, Q10)

In category QC_4, the high selectivity of the text nodes ensures that, even though the base data must be evaluated (to access the text values), the text values have high selectivity leading to few node comparisons. For queries in QC_4, the BranchClassIndex performed better than the PathBasedIndex and MonetDB/XQuery, even after the classification of text values (see Figure 7.10).

Single Step Path Fragment Queries

For single path fragment queries, the BranchClassIndex is optimised because the low cardinality text value (‘*40*’) in query Q08 was classified. Again, MonetDB/XQuery was the most efficient approach in this category in the earlier evaluation, but now the BranchClassIndex is more efficient (BCI_3, BCI_4, BCI_5, BCI_6).

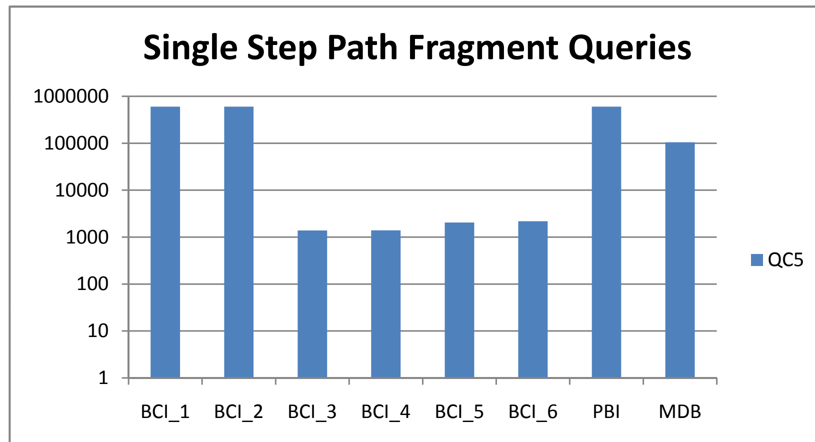


Figure 7.11: Average Single Step Path Fragment Queries

7.3.3 Overall Query Performance

In Table 7.4 the average (AVG) query response time (milliseconds) is shown for each approach. The bar chart in Figure 7.12 illustrates these average times. The BranchClassIndex now performs best overall when a large (BCI_3, BCI_4) or a small (BCI_5, BCI_6) subset of low selectivity text values are indexed.

7.4 Comparison Using Standard Benchmarks

In this section, we evaluate the performance of each approach using the XPathMark benchmark [67], the Computer Science Bibliography [21] and the Protein Sequence Database [58]. In the following experiments, the

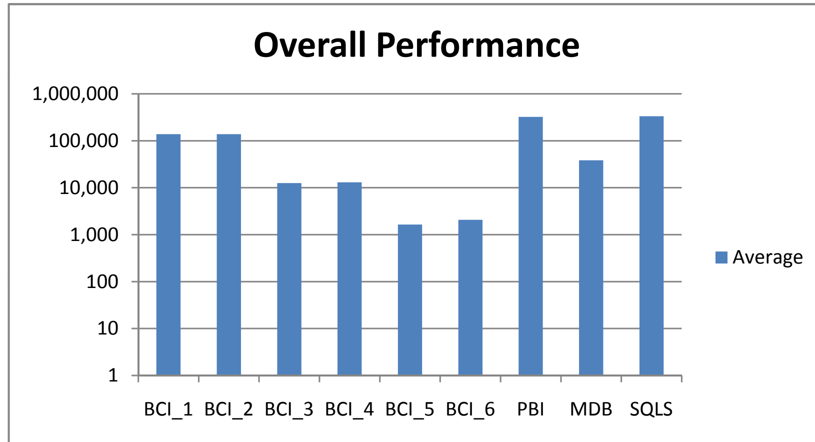


Figure 7.12: Average Performance across all Query Categories

BranchClassIndex is evaluated (1) without text values and *including* the Parent and Child ClassChain components (BCI_1), (2) without text values and *not including* the Parent and Child ClassChain components (BCI_2), and (3) including all of the text values identified as suitable for classification (BCI_2).

No.	Cat	XPath Query	Nodes
Q13	QC_1	/site/regions/africa	1
Q14	QC_4	/site/people/person[@id = 'person0']	1
Q15	QC_1	//regions/africa//item/name	6,600
Q16	QC_2	//person[profile/@income]/name	153,539
Q17	QC_2	//people/person[profile/gender][profile/age]/name	38,583
Q18	QC_1	/site//keyword/ancestor::listitem/text/keyword	373,260
Q19	QC_1	/site/closed_auctions/closed_auction//keyword	15,0047
Q20	QC_2	/site/closed_auctions/closed_auction[./descendant::keyword]/date	64,133
Q21	QC_1	/site/closed_auctions/closed_auction/annotation/description/text/keyword	48,632
Q22	QC_2	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date	31,773
Q23	QC_2	/site//closed_auction[annotation//text//keyword]/date	64,133
Q24	QC_2	/site//item[.//description//listitem]	75,940
Q25	QC_2	/site//item[.//description//listitem]//mailbox//text	72,748

Table 7.6: XMark Queries

7.4.1 The XPathMark Benchmark

The XPathMark benchmark has been widely used by the research community to evaluate the performance of XPath processors [2, 9, 24, 54]. The XMark [67] dataset is used in XPathMark and because it contains synthetic

Query	Cat	BCI_1	BCI_2	BCI_3	PBI	MDB	SQLS
Q13	QC_1	3ms	2ms	12ms	12ms	120ms	1ms
Q15	QC_1	470ms	3,628ms	713ms	21ms	128ms	147ms
Q18	QC_1	3,144ms	4,431ms	8,925ms	>10mins	9,620ms	error
Q19	QC_1	322ms	769ms	464ms	153ms	622ms	582ms
Q21	QC_1	548ms	1,590ms	829ms	47ms	623ms	10ms
AVG		**897ms**	**2,084ms**	**2,188ms**	**120,046ms**	**2,222ms**	**185ms**
Q16	QC_2	1,007ms	1,838ms	3,879ms	>10mins	1,162ms	1,235ms
Q17	QC_2	3,389ms	11,762ms	60,573ms	>10mins	1,131ms	973ms
Q20	QC_2	4,018ms	8,767ms	6,414ms	>10mins	649ms	2,754ms
Q22	QC_2	2,706ms	22,320ms	3,886ms	>10mins	663ms	271ms
Q23	QC_2	1,419ms	2,403ms	3,991ms	>10mins	672ms	>10mins
Q24	QC_2	1,889ms	1,873ms	3,311ms	>10mins	1,516ms	>10mins
Q25	QC_2	4,291ms	4,259ms	7,880ms	>10mins	1,558ms	>10mins
AVG		**2,674ms**	**7,603ms**	**12,848ms**	**600,000ms**	**1,050ms**	**257,890ms**
Q14	QC_4	**168ms**	**159ms**	**357ms**	**235ms**	**132ms**	**118ms**
Total AVG		**1,798ms**	**4,907ms**	**7,787ms**	**369,266ms**	**1,430ms**	**150,507ms**

Table 7.7: Results for the Mark Queries

data it can be generated to any size. We chose an XMark dataset of 1.33 GB in size because it is the largest size XMark dataset that could be stored as an *xml type* in SQL Server.

In Table 7.6, queries Q15-Q22 are taken directly from XPathMark; queries Q13 and Q14 are taken from [31] (which is the node-based approach we evaluate later in §7.5); and finally, we added queries Q23, Q24, and Q25 to illustrate some issues. Within the XMark dataset, a total of 132 text values where identified as suitable for classification all of which are indexed in BCI_3.

For linear path queries (QC_1), SQL Server performed best overall (see Table 7.13). We attribute this to SQL's `PATH` index which is highly optimised for non-branching queries. The BranchClassIndex outperformed the PathBasedIndex for linear path queries because query Q18 took the PathBasedIndex more than ten minutes. The reason for this is that the *ancestor* axis in step three splits the query into three separate primary path fragments and inefficient node comparisons are required between each pair. This shows that the PathBasedIndex can perform poorly for queries in category QC_1 that consist of more than one primary path fragment (an issue that was not

highlighted in the bicycle rental experiments). The BranchClassIndex can always exploit the NCLTV covering index for all but the rightmost step of any linear path query, and therefore outperforms the PathBasedIndex. As with the bicycle rental dataset, the BranchBasedIndex outperforms MonetDB/XQuery for linear path queries, which suggests that it bypasses a larger number of nodes during query processing.

NAME	Nodes	Branch Classes	NCLTV	PC_REL	AD_REL
BCI_1/2	24,645,234	1,337,193	1,653,814	2,676,477	7,459,066
BCI_3	24,645,234	3,097,821	3,817,329	6,198,038	14,403,306

Table 7.8: BranchClassIndex Statistics (XMark)

MonetDB/XQuery performed better than the BranchClassIndex for Twig queries that do not evaluate text values QC_2, whereas in the bicycle rental evaluation, the BranchClassIndex was the most efficient approach. We believe that the reason for this is that, for the XMark dataset, the BranchClassIndex contains a larger number of relationships (see Table 7.8) than it did for the bicycle rental dataset (see Table 7.5). The BranchClassIndex is the second most efficient approach. The PathBasedIndex does not perform well for Twig queries (unless they have high selectivity). This is because costly joins are performed between each branching path (primary path fragment).

SQL Server also did not perform well overall in category QC_2, but it did perform well in queries Q16, Q17, Q20 and Q22. We added queries Q23, Q24, and Q25 (in addition to those in the XPathMark benchmark) to highlight a limitation of SQL Server. Notice that Q23 is a slight variation of Q22 (the predicate contains multiple *ancestor-descendant* relationships rather than *parent-child* relationships). However, the query response time for Q22 is 271ms, whereas Q23 took longer than ten minutes. This suggests that SQL Server's PATH index is efficient for paths that contain multiple parent-child relationships, but not for ancestor-descendant relationships. Queries Q24

and Q25 illustrate this point further.

For Q14 (the only query in QC_4), all approaches took less than 1 second to return the result. This shows again that all approaches performed well for Twig queries that have high selectivity (QC_1). Across all query categories, MonetDB/XQuery is the most efficient approach, followed closely by the BranchClassIndex (368ms less efficient overall). SQL Server and the PathBasedIndex are some orders of magnitude less efficient than MonetDB/XQuery and the BranchClassIndex. Finally, SQL Server is more than twice as efficient as the PathBasedIndex.

BranchClassIndex Evaluation

Indexing text values incurs an overhead for queries (BCI_3 is four time slower overall than BCI_1). The reason for this is the increased index traversal costs for BCI_3 as the index itself is larger than BCI_2 (see Table 7.8).

The inclusion of the Parent and Child ClassChain components has a clear benefit (BCI_1 is almost three times more efficient then BCI_2). This is because the `PC_REL` relation contains significantly fewer relationships than the `AD_REL` relation (see Table 7.8). One factor that increases the number of ancestor/descendant relationships is the *nested-depth*. For example, the XMark dataset has 13 levels, whereas the bicycle rental has only 8. Another factor that can impact the number of ancestor/descendant relationships is text value classification (as pointed out in the bicycle rental evaluation).

7.4.2 The Computer Science Bibliography

Queries executed across the Computer Science Bibliography are mostly taken from other published works in the area of XPath query optimisation [4, 13, 46, 47]. SQL Server 2008 is not included in this evaluation because it had difficulty loading an XML file that has an associated DTD (Document Type Descriptor). A total of 80 text values were identified as suitable for classifi-

No.	Cat	XPath Query	Nodes
Q26	QC_3	/dblp/article[year='1991']/@key	8,126
Q27	QC_1	/dblp/article/author	1,235,495
Q28	QC_1	/dblp/article//sub	3,868
Q29	QC_4	/dblp/article/title[sub='2']	590
Q30	QC_1	/dblp/inproceedings//booktitle	805,306
Q31	QC_5	//sub/ancestor::inproceedings	698
Q32	QC_2	/dblp/inproceedings//title[.//i]//sub	176
Q33	QC_4	/dblp/inproceedings[title='Semantic Analysis Patterns.']/author	2

Table 7.9: Computer Science Bibliography Queries

Query	Cat	BCI_1	BCI_2	BCI_3	PBI	MDB
Q27	QC_1	762ms	782ms	876	1,048ms	3,542ms
Q28	QC_1	13ms	13ms	78	16ms	1,230ms
Q30	QC_1	194ms	192ms	236	669ms	1,837ms
AVG		**323ms**	**329ms**	**397ms**	**577ms**	**2,203ms**
Q32	QC_2	**18ms**	**18ms**	**95ms**	**423,647ms**	**1,943ms**
Q26	QC_3	**326,457ms**	**326,457ms**	**157ms**	**>10mins**	**13,127ms**
Q29	QC_4	540ms	536ms	673	>10mins	8,824ms
Q33	QC_4	863ms	857ms	823	3,007ms	17,645ms
AVG		**702ms**	**697ms**	**748ms**	**301,503ms**	**13,235ms**
Q31	QC_5	**107ms**	**106ms**	**139ms**	**494,903ms**	**9,084ms**
Total AVG		**41,119ms**	**41,120ms**	**385ms**	**265,411ms**	**7,154ms**

Table 7.10: Results for the Computer Science Bibliography

cation all of which are included in BCI_3.

NAME	Nodes	Branch Classes	NCLTV	PC_REL	AD_REL
BCI_1/2	21,228,286	5,984	7,714	11,301	11,731
BCI_3	21,228,286	140,306	173,928	265,791	268,649

Table 7.11: Branch Index Statistics (DBLP)

For linear path queries (QC_1), the BranchClassIndex performed best overall, followed by the PathBasedIndex and then MonetDB/XQuery (see Table 7.10). Thus, again in this category, MonetDB/XQuery's Staircase join performs poorly when compared to approaches that exploit structured information for query processing.

For Twig queries without text values (QC_2), the BranchClassIndex performs best overall followed by MonetDB/XQuery and then the PathBasedIndex.

The PathBasedIndex performed three (costly) joins as Q32 (the only query in this category) consists of three primary path fragments. It is our understanding that the join between steps two and three has a significant performance overhead for MonetDB/XQuery as there are a large number of *inproceedings* and *title* nodes in the computer science bibliography.

Queries in category QC_3 are worst case queries for the BranchClassIndex when text values are not classified. The benefit of classifying text values that have low selectivity is clear as BCI_3 performs 2,079 times faster than BCI_1 and BCI_2. In addition, for the computer science bibliography, there is little performance overhead associated with classifying text values. This is because, in spite of the fact that there is an increase in the size of the BranchClassIndex after the inclusion of text values (see BCI_3 Table 7.11), there are still fewer than 300,000 relationships in the `PC_REL` and `AD_REL` relations.

Queries that have high selectivity (QC_4) show some interesting results. The BranchClassIndex outperforms MonetDB/XQuery and the PathBasedIndex considerably. However, we expected much better results from both these approaches as queries in QC_4 are predicated in text values that have high selectivity. Thus, *join ordering* or poor *query execution plans* may be responsible for these inefficiencies.

The only query in QC_5 is Q31. The BranchClassIndex is the most efficient approach (both before and after text values are indexed) followed by MonetDB/XQuery. In Q31, the ancestor axis at step two ensures that each primary path fragment for the PathBranchIndex approach only spans a single step. Therefore, the PathBaseIndex reduces to the performance of a node-based approach such as the XPath Accelerator.

The BranchClassIndex performs best overall across all categories when the index contains all identified text values and for the computer science bibliography, there is little overhead associated with including the text values

across all categories. MonetDB/XQuery is second followed by the PathBasedIndex. The Parent and Child ClassChain components do not show a performance benefit for this dataset.

No.	Cat	XPath Query	Nodes
Q34	QC_1	//reference/refinfo//author	5,668,287
Q35	QC_1	//ProteinEntry//accinfo/xrefs	281,246
Q36	QC_1	/ProteinDatabase/ProteinEntry//protein//name	262,525
Q37	QC_1	/ProteinDatabase/ProteinEntry//protein//alt-name	42,615
Q38	QC_1	/ProteinDatabase/ProteinEntry//year/ancestor::refinfo	314,763
Q39	QC_1	/ProteinDatabase/ProteinEntry//reference//refinfo//year	314,763
Q40	QC_1	/ProteinDatabase/ProteinEntry//year/ancestor::reference	314,763
Q41	QC_1	/ProteinDatabase/ProteinEntry//classification//superfamily	186,700
Q42	QC_1	/ProteinDatabase/ProteinEntry//reference/refinfo/authors/author	5,668,287
Q43	QC_4	/ProteinDatabase/ProteinEntry[reference/accinfo/accession = 'AE0077']	1
Q44	QC_2	/ProteinDatabase/ProteinEntry[reference//accession]	262,525
Q45	QC_4	/ProteinDatabase/ProteinEntry[reference/refinfo/authors/author = 'Massung, R.F.']	404
Q46	QC_2	/ProteinDatabase/ProteinEntry[reference/refinfo/authors/author]	258,216
Q47	QC_4	/ProteinDatabase/ProteinEntry[organism/variety= 'strain Marburg']/reference/accinfo/xrefs	7
Q48	QC_2	/ProteinDatabase/ProteinEntry[organism/variety]/reference/accinfo/xrefs	9,177
Q49	QC_2	/ProteinDatabase/ProteinEntry[reference//note]	32,107
Q50	QC_2	/ProteinDatabase/ProteinEntry[reference/accinfo/note]	27,429
Q51	QC_3	/ProteinDatabase/ProteinEntry[reference/refinfo/year= '1988']/reference /accinfo[status= 'preliminary']/xrefs	1,477

Table 7.12: Protein Sequence Queries

7.4.3 The Protein Sequence Database

The queries executed across the Protein Sequence Database are largely taken from the experiments presented in [25]. The text node identification algorithms identified a total of 243 text values as being suitable for classification, all of which are indexed in BCI_3.

For linear path queries (QC_1), SQL Server performs marginally better than the BranchClassIndex and MonetDB/XQuery respectively. The PathBasedIndex took more than 10 minutes to evaluate queries Q38 and Q40. This is because they both contain the ancestor axis, which splits each of these queries into two primary path fragments and costly node comparisons are required between each pair. As with the bicycle rental and XMark datasets, the classification of text values has a performance overhead for linear path queries (BCI_1 performs better than BCI_3).

For Twig queries that do not contain text values (QC_2), the BranchClassIndex performs best overall because all but the rightmost step can be evaluated

Query	Cat	BCI_1	BCI_2	BCI_3	PBI	MDB	SQLS
Q34	QC_1	3,232ms	3,382ms	20,852ms	6,274ms	3,945ms	14,924ms
Q35	QC_1	1,896ms	2,262ms	8,590ms	230ms	3,824ms	485ms
Q36	QC_1	1,260ms	1,268ms	11,789ms	222ms	3,780ms	658ms
Q37	QC_1	1,150ms	1,216ms	5,491ms	72ms	3,785ms	193ms
Q38	QC_1	2,196ms	2,276ms	14,215ms	>10mins	4,095ms	error
Q39	QC_1	3,925ms	3,882ms	29,520ms	261ms	3,877ms	618ms
Q40	QC_1	2,066ms	2,100ms	10,100ms	>10mins	4,111ms	error
Q41	QC_1	1,072ms	1,118ms	4,250ms	159ms	3,789ms	344ms
Q42	QC_1	14,215ms	13,320ms	255,703ms	6,367ms	4,286ms	4,337ms
AVG		**3,445ms**	**3,425ms**	**40,057ms**	**134,843ms**	**3,944ms**	**3,080ms**
Q44	QC_2	1,751ms	2,051ms	18,112ms	>10mins	3,942ms	>10mins
Q46	QC_2	5,557ms	7,156ms	117,920ms	>10mins	5,613ms	5,464ms
Q48	QC_2	996ms	1,800ms	1,362ms	>10mins	4,057ms	286ms
Q50	QC_2	1,494ms	1,466ms	3,866ms	>10mins	4,087ms	2,581ms
Q49	QC_2	1,170ms	1,220ms	4,990ms	>10mins	3,959ms	>10mins
AVG		**2,794ms**	**2,739ms**	**29,250ms**	**600,000ms**	**4,332ms**	**241,666ms**
Q51	QC_3	12,102ms	22,525ms	4,054ms	>10mins	27,527ms	379,366ms
Q43	QC_4	61ms	61ms	692ms	136ms	14,191ms	>10mins
Q45	QC_4	185ms	190ms	725ms	10,156ms	21,487ms	324,673ms
Q47	QC_4	96ms	100ms	961ms	4,413ms	18,596ms	>10mins
AVG		**114ms**	**117ms**	**793ms**	**4,902ms**	**18,091ms**	**508,224ms**
Total AVG		**10,429ms**	**11,799ms**	**15,833ms**	**301,334ms**	**5,507ms**	**176,429ms**

Table 7.13: Results for the Protein Sequence Database

NAME	Nodes	Branch Classes	NCLTV	PC_REL	AD_REL
BCI_1/2	22,358,584	2,352,767	2,648,893	4,667,447	8,309,456
BCI_3	22,358,584	5,476,253	6,071,961	10,867,244	21,003,496

Table 7.14: Branch Index Statistics (Protein)

in `NCLTV` index. The PathBasedIndex and SQL Server have to perform a join at each path fragment and do not perform well in this category because there is at least one path fragment for each branching path (i.e. predicate filter) in the query. MonetDB/XQuery shows the second best performance in this category. This again indicates that when the PathBasedIndex or SQL Server must perform joins, MonetDB/XQuery's Staircase Join will perform them more efficiently even though it may perform more of them (one at each step rather than one between each path fragment pair).

Query Q51 is only one query that uses low selectivity text values (QC_3). For the first time, the BranchClassIndex performed best in this category (before text value classification); MonetDB/XQuery performed best in all previous

cases. We believe that there are two reasons for this. Firstly, even though the text value '*1998*' is not indexed, it only occurs 25,773 times, which limits the number of node comparisons. Secondly, there are nine steps in Q51 (the most out of all queries considered here) and MonetDB/XQuery must perform a Staircase join between each pair. As previously, the PathBasedIndex and SQL Server perform at the same level in this category because of the large number of path fragments. However, query Q51 is three times faster after text values have been classified (BCI_3). The Parent and Child ClassChain components result in increased performance for query Q51 because of the large number of child axes and the fact that the `PC_REL` relation contains half as many relationships as the `AD_REL` relation (see Table 7.14).

For queries that evaluate high selectivity text values (QC_4), SQL Server took more than ten minutes to return a result for Q43 and Q47. On all previous experiments, SQL Server performed well in this category. The reason for this is unclear to us, but it may be related to join ordering. The BranchClassIndex performed best overall, which suggests that the query optimiser chooses better execution plans than it does for the PathBasedIndex. For queries with high selectivity, the classification of text nodes and the inclusion of the Parent and Child ClassChain components show little benefit (BCI_1, BCI_2, and BCI_3 perform at the same level).

Overall MonetDB/XQuery performs best, followed by the BranchClassIndex (before text values are indexed (BCI_1)). This suggests that the inclusion of *all* 243 text values (BCI_3) is not optimal. However, the classification of text values significantly increases the performance of queries in QC_3 (Q51 was reduced from 12 seconds to 4 seconds). Comparatively, SQL Server and the PathBasedIndex are inefficient for the protein sequence dataset.

7.5 Node Based Approaches

In this section, we will evaluate the performance of a traditional node-based approach to XPath optimisation [31], which is an optimised version of the XPath accelerator [29]. Additionally, an evaluation of the node partitioning approach that is most similar to ours [43] is provided. We will refer to [31] as the NodeApproach and [43] as the PartitionApproach.

For the NodeApproach, we built the following three *partitioned B-tree* indexes as suggested in [31]: *Node(level,pre)*, *Node(type,name,pre)* and *Node(type,name,level,pre)*. Additionally we built indexes on *size*, *name*, *level*, *value* and *type*. For the PartitionApproach, we used partitioning factors 20, 40, 60, and 100. As suggested in [43], *Node(pre)* is a primary key; *Node(part)* is a foreign key reference to the primary key *Part(part)*; and indexes were built on *Node(post)*, *Node(name)*, *Node(part)*, *Part(pre)*, and *Part(post)*.

Our overall finding for both approaches is that they do not scale well, even for relatively small XML documents. As such, we had to evaluate these approaches using a relatively small dataset. Thus, for the following experiments, we generated an XMark dataset of just 115 MB in size and tested both approaches against queries from the *XPathMark* [67] benchmark and some queries taken from [31] (these queries are shown in Table 7.6).

Query	**Q13**	**Q14**	**Q15**	**Q16**	**Q17**	**Q18**	**Q19**	**Q20**	**Q21**	**Q22**
PartitionApproach(20)	211	223	>10mins	>10mins	53,481	>10mins	5,198	>10mins	126,190	151,728
PartitionApproach(40)	263	307	>10mins	>10mins	61,458	>10mins	9,168	>10mins	197,386	140,133
PartitionApproach(60)	260	1,452	>10mins	>10mins	52,423	>10mins	10,492	>10mins	124,019	132,178
PartitionApproach(80)	262	1,200	>10mins	>10mins	78,719	>10mins	10,215	21,281	114,020	161,539
PartitionApproach(100)	267	1,134	>10mins	259,528	53,596	>10mins	290,967	18,289	112,605	166,413
NodeAppraoch	136	259	>10mins	>10mins	>10mins	>10mins	23,842	>10mins	>10mins	>10mins
BranchClassIndex	16	92	63	81	3,274	896	229	1,371	192	996

Table 7.15: Results for the Node Based Approaches

In Table 7.15, the query response time for each of these queries is shown. These results show the following:

- The NodeApproach timed out on all but Q13, Q14, Q18.
- In the PartitionApproach, a partitioning factor of 100 returned re-

sults for the greatest number of queries (Q13, Q14, Q16, Q17, Q19, Q20, Q21, Q22). Query Q19 shows an increase in processing times as the partitioning factor increased, whereas Q21 shows a decrease. The remaining results do not suggest such a pattern.

- The PartitionApproach returned results for a greater number of queries then the NodeApproach across all partitioning factors.

- The BranchClassIndex is orders of magnitude faster across all queries.

Queries Q13 and Q14 have *high selectivity* as they return a single result node. Also, the first two steps in Q19, that is, `/site` and `/closed_auctions`, both evaluate just a single node only. We attribute the fact that NodeApproach returned results for queries Q13, Q14 and Q15 to the high selectivity of these queries.

There is no consistent pattern between the incrementing partitioning factors indicating that no single partitioning factor per dataset is ideal. The PartitionApproach provides superior results than the NodeApproach, both in terms of query response times, and in terms of the number of queries that returned a result within 10 minutes. However, the exhaustive experimentation required to identify suitable partition factors is infeasible (in terms of index build times and query analysis). Both approaches do not scale well for queries that have low selectivity, because even for relatively small XML datasets (115 MB), the query response times are large relative to those of the BranchClassIndex.

7.6 Summary

In this chapter, we evaluated the performance of the BranchClassIndex; a path-based indexing approach (PathBasedIndex); a leading open-source (MonetDB/XQuery); and a commercial (SQL Server 2008) XML database.

Linear path queries (QC_1) do not contain predicate filters, therefore all approaches are effective in this category. Across all datasets, SQL Server performed marginally better than the BranchClassIndex, followed by the PathBasedIndex and finally MonetDB/XQuery is the poorest performing approach in this category.

The BranchClassIndex is the most effective approach for queries in QC_2 for all but the XMark dataset. For queries executed across the XMark dataset, MonetDB/XQuery is more efficient. The reason for this is the increased number of ancestor/descendant and parent/child relationships. We suggest ways of minimising this increase to the index size in our discussion on future research (Chapter 8).

For Twig queries that evaluate text values with low selectivity QC_3, the BranchClassIndex performs best for the protein sequence database, whereas MonetDB/XQuery performs best for the bicycle rental and computer science bibliography datasets (the XPathMark evaluation did not contain queries in this category). However, for the bicycle rental dataset, the BranchClassIndex is 162 times faster than MonetDB/XQuery when a small subset of text values are indexed and 12 times faster when a large subset of text values are indexed. For the computer science bibliography, the BranchClassIndex is 83 times faster than MonetDB/XQuery after all low selectivity text values have been indexed.

For queries that evaluate text values that have high selectivity (QC_4), SQL Server performed best for the bicycle rental and XMark datasets, followed by the BranchClassIndex, MonetDB/XQuery and the PathBasedIndex respectively. For the computer science bibliography and protein sequence dataset, the BranchClassIndex is the most efficient approach.

Overall for the bicycle rental dataset and protein sequence datasets, MonetDB/XQuery is the most efficient approach. However, for the bicycle rental dataset, the BranchClassIndex replaces MonetDB/XQuery as the most ef-

ficient approach when a small or a large subset of text values are indexed. The overall averages for the XMark evaluation show that the BranchClassIndex and MonetDB/XQuery perform at the same level, whereas the PathBasedIndex and SQL Server are less efficient by orders of magnitude. For the computer science bibliography, the BranchClassIndex performs best overall when all low selectivity text nodes are indexed.

These experiments show that the BranchClassIndex is a poor choice for one query category (QC_3). This is because queries in this category contain text values that have low selectivity, thus leading to large numbers of node comparisons. However, it was shown that indexing low selectivity text values improves the performance of the BranchClassIndex making it the most efficient approach overall for the bicycle rental and computer science bibliography datasets. For datasets where the BranchClassIndex did not perform best overall (XMark and protein sequence datasets), its storage costs are higher. Therefore, as part of our future work, we discuss ways in which the BranchClassIndex can be compacted.

Chapter 8

Conclusions

This dissertation presented a node partitioning strategy for optimising XPath queries. In this chapter, we restate the concepts that were introduced and discuss future work. This chapter is structured as follows: in §8.1, a summary of the dissertation is provided and in §8.2, the future potential of this work is discussed.

8.1 Thesis Summary

The partitioning approach, which to the best of our knowledge is shared with just one other index-based approach [43], allows nodes of different names and types (element and attribute) to reside in the same partition. The hypothesis is based on the fact that there will always be fewer partitions then there are nodes. Therefore, the partitions that contain target nodes can be identified directly and all other partitions are eliminated from the search space. In contrast to [43], however, our approach allows partitions of different sizes within an XML document and avoids time-consuming preprocessing to identify suitable partitioning factors.

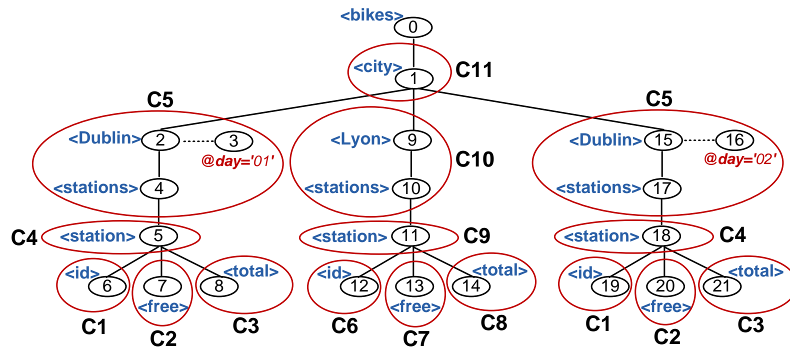
The partitioning approach optimises the most commonly used XPath axes (the six hierarchical axes). We began by defining a disjoint partition of nodes

as a *branch* because suitable partitions are identified based on the layout of *branching nodes* (nodes that have at least two children) within the XML document. It soon became clear that the most suitable partitioning strategy for the hierarchical axes was to ensure that only nodes that are hierarchically related can reside in the same branch. We showed that this has the effect of reducing the number of false hits, thus improving the performance of the hierarchical axes.

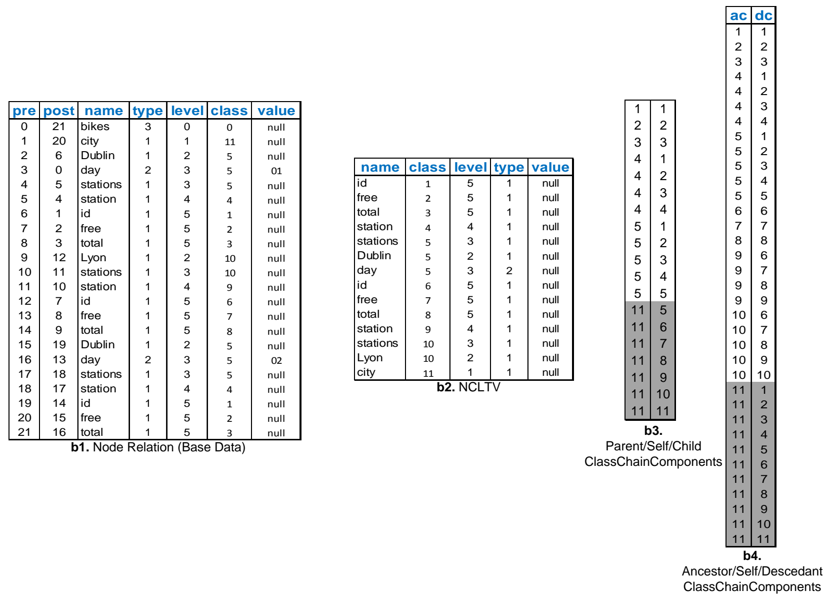
A side effect of using the rule that only hierarchically related nodes can share the same branch is that there may be a large number of branch instances. Thus, because the optimisation strategy is based on the fact that there will be fewer partitions than nodes, the increased number of branch instances may result in a performance overhead. To overcome this issue, we provided a classification process for equivalent branch instances. After classification, a single branch class represents any number of branch instances. In effect, the index is compacted while maintaining its search space pruning benefits. Based on the fact that a single branch class represents many branch instances, we then showed that a single branch class proxy can be processed in place of large numbers of branch instances for many steps in XPath expressions. As a result, we showed that for many XPath expressions the majority of nodes are bypassed during the query process.

After the branch classification process, we described how the index can be deployed in a relational database. We provided a detailed description of the relations that were used in its deployment and detailed an XPath-to-SQL transformation process for the core subset of the XPath language.

Finally, we demonstrated the performance of the index using four large XML datasets. A real world dataset was used to demonstrate the wider impact of our approach. Then, we exploited three datasets that are commonly used by the research community for benchmarking purposes.



(a) XML Snippet from the Bicycle Rental Dataset



pre	post	name	type	level	class	value
0	21	bikes	3	0	0	null
1	20	city	1	1	11	null
2	6	Dublin	1	2	5	null
3	0	day	2	3	5	01
4	5	stations	1	3	5	null
5	4	station	1	4	4	null
6	1	id	1	5	1	null
7	2	free	1	5	2	null
8	3	total	1	5	3	null
9	12	Lyon	1	2	10	null
10	11	stations	1	3	10	null
11	10	station	1	4	9	null
12	7	id	1	5	6	null
13	8	free	1	5	7	null
14	9	total	1	5	8	null
15	19	Dublin	1	2	5	null
16	13	day	2	3	5	02
17	18	stations	1	3	5	null
18	17	station	1	4	4	null
19	14	id	1	5	1	null
20	15	free	1	5	2	null
21	16	total	1	5	3	null

b1. Node Relation (Base Data)

name	class	level	type	value
id	1	5	1	null
free	2	5	1	null
total	3	5	1	null
station	4	4	1	null
stations	5	3	1	null
Dublin	5	2	1	null
day	5	3	2	null
id	6	5	1	null
free	7	5	1	null
total	8	5	1	null
station	9	4	1	null
stations	10	3	1	null
Lyon	10	2	1	null
city	11	1	1	null

b2. NCLTV

1	1
2	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
11	5
11	6
11	7
11	8
11	9
11	10
11	11

b3. Parent/Self/Child ClassChainComponents

ac	dc
1	1
2	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
6	6
7	7
8	8
9	6
9	7
9	8
9	9
10	6
10	7
10	8
10	9
10	10
11	1
11	2
11	3
11	4
11	5
11	6
11	7
11	8
11	9
11	10
11	11

b4. Ancestor/Self/Descedant ClassChainComponents

(b) Relations in the BranchClassIndex

Figure 8.1: Relational Index Deployment Revisited

8.2 Future Work

In this section, we describe the long and short term goal of our on-going research.

8.2.1 Reducing Redundancy

In this dissertation, we showed how the ClassChain components can be stored in relations that are suitable for equijoin evaluations; these relations

are shown in Figure 8.1b3 and 8.1b4. We chose this approach because the author of [41] showed us that relational databases can evaluate equijoins efficiently (unlike non-equijoins).

An issue associated with a relational storage method that is suitable for equijoin evaluation is redundancy (see Chapter 2 for more details). The approach presented in this dissertation reduces redundancy to a large degree through branch classification, but we believe it can be reduced further.

Consider node 1 in Figure 8.1a (*city*). This node's branch class identifier (`C11`) is an ancestor class of all other branch classes in the XML document (except the class that the document node belongs to). Redundancy occurs because `C11`'s descendant classes (highlighted in grey in Figure 8.1b3 and 8.1b4) also occur as descendants of other branch classes. For example, class `C9` is a descendant of class `C11` and `C10`.

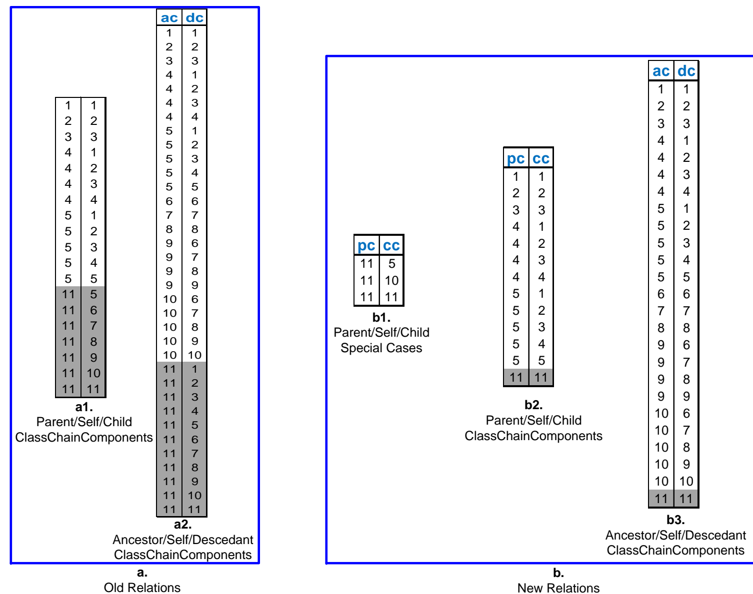


Figure 8.2: Redundancy Reduction

Only branch classes that occur at low levels in the XML document that cause significant redundancy issues. For example, the class for node *city* (level one) in the bicycle rental dataset or *regions* (level one) in the XMark

dataset. We believe that it is possible to identify such branch classes within the XML document and reduce the impact on redundancy for the following reason.

A branch class such as `C11` is an ancestor of classes `C1-C11`. Another way of looking at it is that branch class `C11` is a parent of classes `C5` and `C10`, and is an ancestor of `C5` and `C10`'s descendants. Thus, we propose the new relational layout shown in Figure 8.2. The special case branch classes that are ancestors of a large number of branch classes can be placed in a separate relation as shown for class `C11` in Figure 8.2b1 and the redundant duplications can be removed from 8.2b2 and 8.2b3 as shown. We also propose that this is the approach that will be used to include the document node in future versions of the BranchClassIndex.

The XPath-to-SQL transformation process would need to be updated to perform an additional check across such a new relation. However, we don't foresee this to be a difficult problem, and we project that the number of special case nodes would be small within each XML document, thus the additional check should not be too costly.

8.2.2 Other Future Directions

In this dissertation, we have described an optimisation strategy for the hierarchical XPath axes. We focused on the hierarchical axes because, in our experience, they are the most commonly used axes. However, integration of other axes such as *attribute* should require only simple extensions to the XPath-to-SQL transformation process. In fact, the attribute axis should already benefit from our approach as attribute nodes are placed in the same branch class as their parent element. Thus, identifying an element node's attributes will be the efficient task of identifying the attribute nodes in the same branch class as itself.

In contrast, to date we have not seen any real world requirements for the

following and *preceding* XPath axes. Thus, we propose addressing these axes in the future if we can motivate the problem based on further research. Also, we intend to determine the benefit of our branch partitioning approach for the broader XQuery language.

Finally, our ultimate goal is determine and overcome the overhead of maintaining the BranchClassIndex upon XML updates. The BranchClassIndex is encoding scheme independent, thus it can be used along with more update friendly encoding schemes such as [10, 55]. However, the introduction of new nodes may invalidate branch classes. Alternatively, it could be that new nodes are inserted into already existing branch classes leading to little index maintenance overhead.

Bibliography

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] Loredana Afanasiev, Massimo Franceschet, and Maarten Marx. XCheck: A Platform for Benchmarking XQuery Engines. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1247–1250. VLDB Endowment, 2006.

[3] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, pages 141–, 2002.

[4] Toshiyuki Amagasa, Lianzi Wen, and Hiroyuki Kitagawa. Proximity Search of XML Data Using Ontology and XPath Edit Similarity. In *DEXA*, pages 298–307, 2007.

[5] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree Pattern Query Minimization. *The VLDB Journal*, 11(4):315–331, 2002.

[6] Tim Anderson. The XML Files. In *Personal Computer World*, 2001.

[7] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path Summaries and Path Partitioning in Modern XML Databases. *World Wide Web*, 11:117–151, March 2008.

[8] N. A. Aznauryan, Sergei D. Kuznetsov, L. G. Novak, and M. N. Grinev. SLS: A Numbering Scheme for Large XML Documents. *Programming and Computer Software*, 32(1):8–18, 2006.

[9] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyên. Type-Based XML Projection. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 271–282. VLDB Endowment, 2006.

[10] Timo Bohme and Erhard Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *in RDBMS. Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004*, pages 70–81, 2004.

[11] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: Xquery—the relational way. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1322–1325. VLDB Endowment, 2005.

[12] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2006. ACM.

[13] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic xpath processing in natix. In *ICDE*, pages 705–716, 2005.

[14] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, New York, NY, USA, 2002. ACM.

[15] Artem Chebotko, Mustafa Atay, Shiyong Lu, and Farshad Fotouhi. XML subtree reconstruction from relational storage of XML documents. *Data Knowl. Eng.*, 62(2):199–218, 2007.

[16] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 134–144, New York, NY, USA, 2003. ACM.

[17] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries Over XML Documents. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 283–294. VLDB Endowment, 2006.

[18] Yi Chen, Susan B. Davidson, and Yifeng Zheng. BLAS: An efficient XPath processing system. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 47–58, New York, NY, USA, 2004. ACM.

[19] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 121–132, New York, NY, USA, 2002. ACM.

[20] Michael Coles. *Pro SQL Server 2008 XML*. Apress, 2008.

[21] Computer Science Bibliography. Online Resource. http://dblp.uni-trier.de.

[22] P. Dietz and D. Sleator. Two Algorithms for Maintaining Order in a List. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM.

[23] Andrey Fomichev, Maxim Grinev, and Sergei D. Kuznetsov. Sedna: A Native XML DBMS. In *SOFSEM*, pages 272–281, 2006.

[24] Haris Georgiadis and Vasilis Vassalos. Improving the Efficiency of XPath Execution on Relational Systems. In *EDBT*, pages 570–587, 2006.

[25] Haris Georgiadis and Vasilis Vassalos. XPath on Steroids: Exploiting Relational Engines for XPath Performance. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 317–328, New York, NY, USA, 2007. ACM.

[26] Mark Roantree Gerard Marks and John Murphy. Classification of Index Partitions. Technical report, Dublin City University, 2010. http://www.computing.dcu.ie/[insert *tilde* character]isg/publications/ISG-10-03.pdf.

[27] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.

[28] Gang Gou and Rada Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Trans. on Knowl. and Data Eng (TKDE).*, 19(10):1381–1403, 2007.

[29] Torsten Grust. Accelerating XPath Location Steps. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, New York, NY, USA, 2002. ACM.

[30] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[31] Torsten Grust, Jan Rittinger, and Jens Teubner. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 949–958, New York, NY, USA, 2007. ACM.

[32] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch Its (axis) Steps. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 524–535. VLDB Endowment, 2003.

[33] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11:274–291, 2002. 10.1007/s00778-002-0081-x.

[34] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. Path Materialization Revisited: An Efficient Storage Model for XML Data. In *Proceedings of the 13th Australasian database conference*, pages 85–94, Darlinghurst, Australia, 2002. ACM.

[35] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering Indexes for Branching Path Queries. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144, New York, NY, USA, 2002. ACM.

[36] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *In ICDE*, pages 129–140, 2002.

[37] Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 533–542, New York, NY, USA, 2004. ACM.

[38] Jiaheng Lu, Ting Chen, and Tok Wang Ling. TJFast: Effective Processing of XML Twig Pattern Matching. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1118–1119, New York, NY, USA, 2005. ACM.

[39] Olli Luoma. Indexing xml data with a schema graph. In *Databases and Applications*, pages 274–279, 2004.

[40] Olli Luoma. Modeling nested relationships in xml documents using relational databases. In *SOFSEM*, pages 259–268, 2005.

[41] Olli Luoma. Supporting xpath axes with relational databases using a proxy index. In *XSym*, pages 99–113, 2005.

[42] Olli Luoma. Xeek: An efficient method for supporting xpath evaluation with relational databases. In *ADBIS Research Communications*, 2006.

[43] Olli Luoma. Efficient Queries on XML Data through Partitioning. In *WEBIST (Selected Papers)*, pages 98–108, 2007.

[44] Stefan Manegold. An empirical evaluation of xquery processors. *Inf. Syst.*, 33:203–220, April 2008.

[45] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha. The Repeatability Experiment of SIGMOD 2008. *SIGMOD Rec.*, 37:39–45, March 2008.

[46] Gerard Marks and Mark Roantree. Pattern Based Processing of XPath Queries. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 179–188, New York, NY, USA, 2008. ACM.

[47] Gerard Marks and Mark Roantree. Metamodel-Based Optimisation of XPath Queries. In *BNCOD*, 2009.

[48] Gerard Marks and Mark Roantree. Classification of Index Partitions to boost XML Query Performance. In *29th International Conference on Conceptual Modeling Vancouver, BC, Canada*, 2010.

[49] Gerard Marks, Mark Roantree, and Dominick Smyth. Optimising Queries for Web Generated Sensor Data. In *The 22nd Australasian Database Conference, Perth, Australia, 2011*, 2011. To Appear.

[50] Dónall McCann, Mark Roantree, Niall Moyna, and Michael Whelan. Synchronizing sensed data in team sports. *ERCIM News*, 2009(76), 2009.

[51] W Meier. Index-driven XQuery Processing in the eXist XML Database. In *XML Prague*, (2006).

[52] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 277–295, London, UK, 1999. Springer-Verlag.

[53] Matthias Nicola. Lessons Learned from DB2 pureXML Applications: A Practitioner's Perspective. In *XSym*, pages 88–102, 2010.

[54] Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML Transaction Processing Benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 937–948, New York, NY, USA, 2007. ACM.

[55] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM.

[56] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing xml data stored in a relational database. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1146–1157. VLDB Endowment, 2004.

[57] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML data stored in a relational database. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1146–1157. VLDB Endowment, 2004.

[58] Protein Sequence Database. Online Resource . http://www.cs.washington.edu/research/xmldatasets/.

[59] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make Twig Pattern Matching Fast. In *DASFAA*, pages 850–862, 2007.

[60] Mark Roantree, Colm Noonan, and John Murphy. Specifying and Optimising XML Views. In Richard Cooper and Jessie Kennedy, editors, *Data Management. Data, Data Everywhere*, volume 4587 of *Lecture Notes in Computer Science*, pages 138–146. Springer Berlin / Heidelberg, 2007.

[61] Mark Roantree and Mikko Sallinen. Introduction - The Sensor Web - Bridging the Physical-Digital Divide. *ERCIM News*, 2009(76), 2009.

[62] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[63] J.X. Wong K.-F. Li J. Tang, N. Yu. Fast XML Structural Join Algorithms by Partitioning. *Journal of Research and Practice in Information Technology*, 40:33–54, 2008.

[64] Jens Teubner. The Relational XQuery Puzzle: A Look-Back on the Pieces Found So Far. *Computer Science - R&D*, 24(1-2):37–49, 2009.

[65] Wei Wang, Haifeng Jiang, Hongzhi Wang, Xuemin Lin, Hongjun Lu, and Jianzhong Li. Efficient Processing of XML Path Queries Using the Disk-Based F&B Index. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 145–156. VLDB Endowment, 2005.

[66] XML Path Language 2.0. Online Resource . http://www.w3.org/TR/xpath20.

[67] XPathMark Benchmark. Online Resource. http://sole.dimi.uniud.it/ massimo.franceschet/xpathmark/.

[68] XQuery 1.0 and XPath 2.0 Data Model (XDM). Online Resource . http://www.w3.org/TR/xpath-datamodel.

[69] Masatoshi Yoshikawa and Toshiyuki Amagasa. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Trans. Internet Technol.*, 1(1):110–141, 2001.

[70] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 425–436, New York, NY, USA, 2001. ACM.